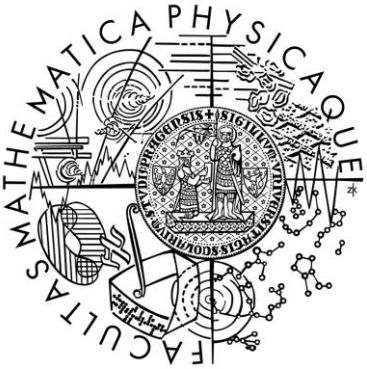
Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS



Petr Čečil

Comparison of commonly used SQL and NoSQL data storages

Department of Software Engineering

Supervisor of the bachelor thesis: Mgr. Martin Nečaský, Ph.D.

Study programme: Computer Science

Specialization: General Computer Science

Prague 2011

I would like to thank Martin Nečaský for his supervising and for his time, ideas and advices.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague date2011/5/26 signature

Název práce: Porovnání běžných SQL a NoSQL datových úložišt

Autor: Petr Čečil

Katedra / Ústav: Katedra softwarového inženýrství

Vedoucí bakalářské práce: Mgr. Martin Nečaský, Ph.D.

Abstrakt:Ukládání dat je dnes velmi důležité téma. Kvůli Web 2.0 a software-as-a-service aplikacím vzniká roztoucí potřeba škálovatelnosti a hledání nových typů datových úložišť.

Cílem této práce je pomoci pochopit konkurenční datová úložiště SQL a NoSQL a jejich vhodnost užití. Autor zmapoval poslední trendy v ukládání dat a aplikační architektuře a snažil se zjistit, jak je konkrétní databáze řeší. Součástí práce je také experimentální část s jednoduchou aplikaci, která demonstruje konektory jednotlivých databází a jejich rychlost.

Klíčová slova: SQL, NoSQL, databáze

Title: Comparison of commonly used SQL and NoSQL data storages

Author: Petr Čečil

Department / Institute: Department of Software Engineering

Supervisor of the bachelor thesis: Mgr. Martin Nečaský, Ph.D.

Abstract: Data storing is today very important topic. Because of Web 2.0 and software-as-a-service applications there is growing need for scalability and new types of data stores.

The aim of this thesis is to help understand competing SQL and NoSQL data stores and their target use cases. Author mapped last trends in data storing and application architecture and tried to find how concrete data stores address them. There is also experimental part with benchmark and simple application that demonstrate data store’s connectors and their speed.

Keywords: SQL, NoSQL, databases

Contents

Introduction

Today, we are taking a hard look at the way our applications store and retrieve data, and we are asking a question: Do we really need traditional RDBMS[1] for all scenarios? This does not mean throwing away all relational databases, it means not using one-database-fits-all approach. We are trying to find best tools for our job.

So my aim in this thesis is to help us understand competing SQL/NoSQL (for definitions see chapters 2 and 3) data stores so that we are best armed to make the right choice of database for our needs.

In chapter number 1, there is a description of latest trends in data storing, size and application architecture. Chapter 2 shows few common SQL databases and describes their limitations. Chapter 3 introduces some ready to use NoSQL data stores and how they address applications needs. Experimental part is in chapter 4.

[1]RDBMS – is a software package to control use, maintenance and creation of relational database

1. Challenges

In this chapter, I want to describe latest trends in application development and data storing. Their set will help us specify the requirements for comparing database solutions in next chapters.

1.1. Data Set Size

Internet scales. There is a report from research company IDC [1] that attempt to count up all digital data created each year. It says that for year 2006 there were 161 exabytes of digital data. But in 2010, it is expected that total amount will jump to 988 exabytes (yes we are closing in on 1 zettabyte).

They predict exponential growth, over two years more data will be generated than during all the previous years combined. Today, we have massive data volumes with distributed architecture required to store the data – Google, Amazon, Yahoo, Facebook (e.g. 10-100K servers). We are talking here about massive data collections, coming 24/7 from geographic areas all around the globe. This is a huge explosion in the growth of information.

1.1.1. UPDATEs and DELETEs

Usually we do not really need UPDATEs and DELETEs because they lead to loss of information. We may need the data later for reactivation or auditing. Typically information is never removed nor just updated from a real world perspective anyway. E.g. user leaves company - his employment record is still saved, or account balance is updated - previous record is maintained. So we can typically model an UPDATE/DELETE as an INSERT and version the record.

But some problems arise when we use INSERT-only system. When versioned data gets too large we archive inactive parts e.g. on different machine. Database cannot

help us with cascading[2]; this needs to be done on app layer. Also cascades can be far more complex than propagating UPDATE/DELETE. E.g. every time bank account is debited checks need to be made on minimum account balance, etc... Also queries will need to filter out inactive records but we can use views to help cache that.

1.1.2. Scaling

As database fills with data, we must be prepared for scaling; otherwise it will stop serving its purpose – saving our data. There are two approaches for database scaling.

Scaling Up

Scaling up is the most common approach for scaling applications, because it requires no change to application and keeps all data in one place which helps reducing the maintenance and complexity. Sometimes it is also called Vertical Scaling. It simply means increasing resources for application (adding faster processors, more memory and upgrading discs), allowing it to handle higher load and amount of requests.

Unfortunately, the cost of this approach does not scale nearly linearly – getting a machine that can support twice as many discs can cost more than twice as much. Also this approach helps us only for some time. Our box will be maxed out and in the end only option, we have left is Scaling Out.

Scaling Out

Scaling out is scaling by division, also called Horizontal Scaling, Partitioning or Sharding. This is more complex approach, which is based on partitioning data across several database servers. This allows distributing load at more machines.

There are two different types of solution to this problem depicted on Figure 1.

[2]Cascading – it‘s a refential action defined e.g. in SQL:2003 revision of SQL language, when the column is updated or deleted same action will happen on referenced column

Horizontal partitioning

As the title implies it involves splitting table by rows into different tables. Each table then contains the same number of columns but fewer rows. For example, we can split table to 12 by every month in year while a view with a union might be created over all of them to provide a complete view of all rows.

Vertical partitioning

Vertical partitioning means splitting table by columns into different tables. Normalization also splits tables by columns but vertical partitioning goes above that and can splits tables that are already normalized. We can also use different machines to store infrequently or very wide columns. Common way to split table is by static (fast to find) and dynamic (slow to find) data, and therefore gaining performance boost in accessing static data e.g. for statistical analysis.

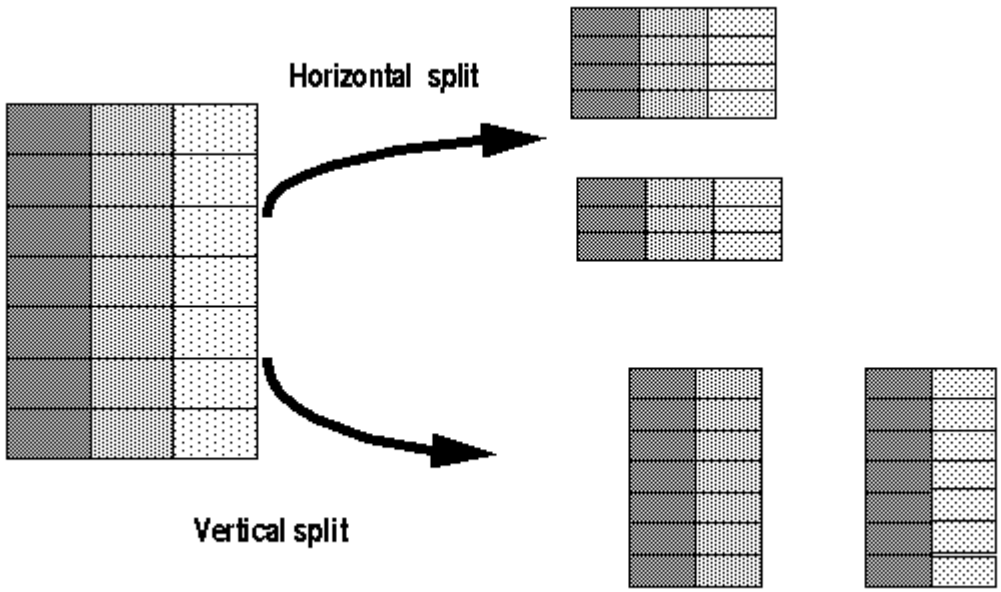


Figure 1 – the difference between Horizontal and Vertical partitioning [2]

1.1.3. Replication

With many distributed data solutions (such as MySQL clusters), we can set up multiple copies of data on different servers in process called replication.

It is a process of sharing data between redundant databases (with ensuring consistency). It helps to improve fault-tolerance, reliability and accessibility. Actually, it is scaling by replication. They all can serve simultaneous requests and improve performance by that.

There are two types of replication.

Master-Slave

Typically process of replication is not decentralized, but performed by Master-Slave relationship. All servers do not work the same way. The master is an authoritative source of all data and slave nodes synchronize their copies with him.

There are some obvious disadvantages. Each write results in N writes on N slaves. This can pose a problem when we deal with high volume of data and can also limit size of scaling. Although read can get faster (because we can read from N nodes) but critical reads still must go to the master because writes may not be propagated to all nodes. Usually this needs to be implemented on application layer. The main problem is that the master node is a bottle-neck for whole database (we cannot write faster than master node is capable) and also single point of failure (when master node fails e.g. not all of his data can be propagated to slaves so everyone will have different data).

Master-Master

When all of the nodes have exactly the same function and there is no special host that is coordinating activities, we call it Master-Master replication (sometimes also called "server symmetry").

More masters improve write scalability but may lead to conflicts. Conflict resolution happens at $O(N^3)$ or $O(N^2)$ *[3]*. Decentralized design is also key for high availability, failure of one node will not distrupt service. Also setting up one node does not differ from setting up 50 nodes, because it is usually the same – scaling is linear.

1.2. Connectedness of Information

Over time, information became more and more interlinked and connected. Starting with documents - completely isolated information structures, hypertext added links to documents, blogs has pingbacks (basically request from blog A to blog B, send when one blog links to another), tagging groups etc...

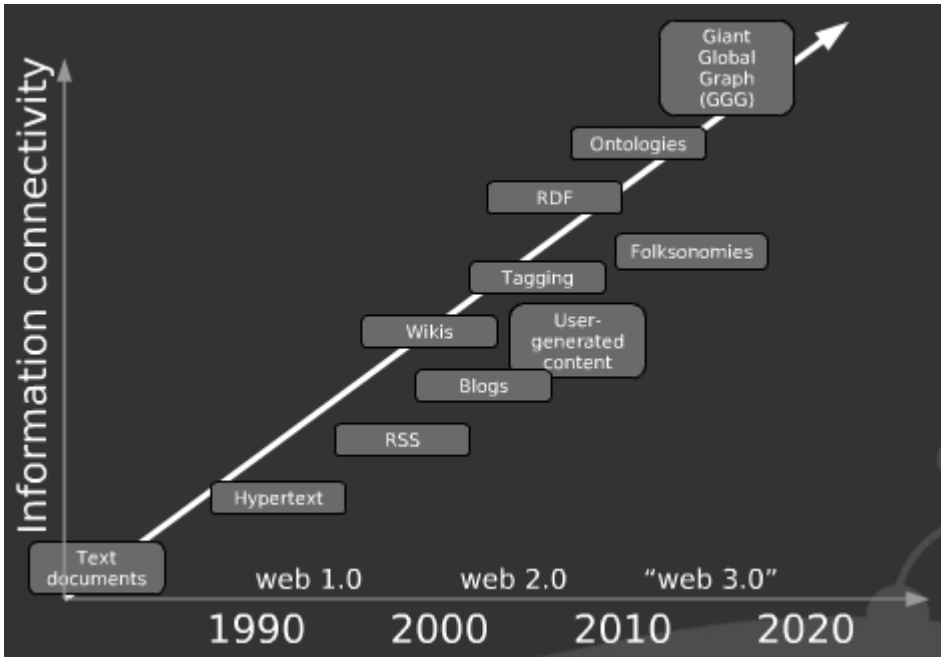


Figure 2 – example of information connectivity [4]

On Figure 2 we can see growth of information connectivity through time, with some examples.

1.2.1. JOINs

JOIN is syntactic clause in SQL language used for combining multiple tables in database. In general, it is used to describe process of connecting multiple sets of data (not only in SQL databases).

We should avoid them because they do not work across shards, cached views are still not supported by majority of databases and most importantly: joins are expensive on performance because database server must perform complex operations over large volumes of data.

How to avoid them? By denormalization. Purpose of normalization is in reducing amount of storage and in making easier to have consistent data by keeping just one copy. But when we denormalize, we will have to ensure consistency at application layer. Also data volume will start to grow but storage is cheap today and we can archive not-so-much-used data. And it is quite easy when we do only INSERTs and no UPDATEs and DELETEs.

1.3. Semi-Structured Information

We can define Semi-Structured information as information that has few mandatory parts but many optional parts. For example Salary lists: in 1980s all elements had exactly one job, but in 2000s we need 5 job columns, or 6? Or 12? Content individualizes.

We can look at RDBMS performance in this area at Figure 3:

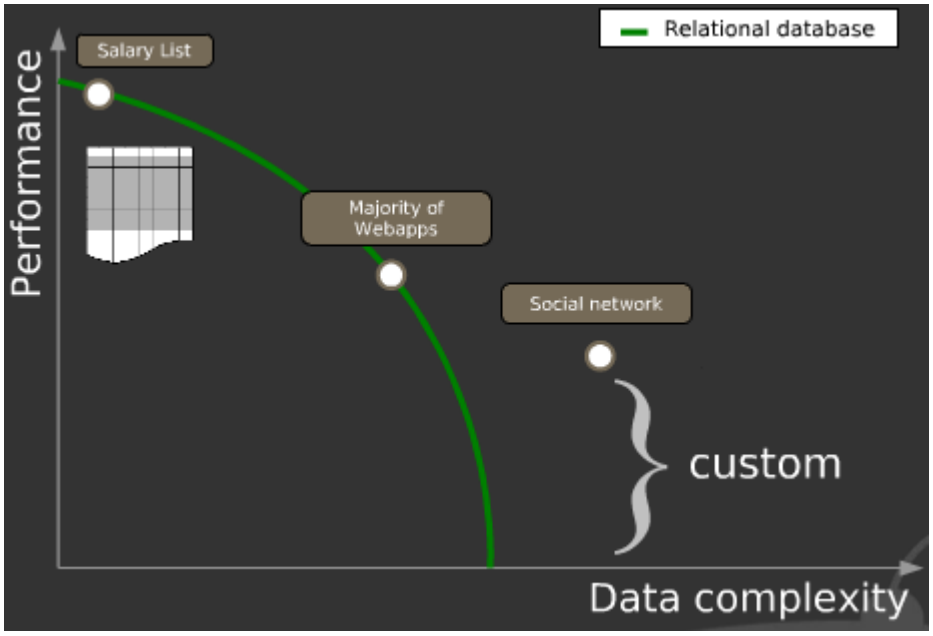


Figure 3 – Performance of RDBMS compared to data complexity [4]

We joined connectedness and semi structure into Data complexity in this graph. Points are requirements of applications.

Simple data means high performance (e.g. salary lists) and simple tables are best target use case for relational databases. Due to growth of complexity followed by using of sparse tables (case is represented as multiple rows, with each row

representing a property/value pair, for e.g.: CustomerID, ProductID, Quantity) and JOINs we can see dropping performance. Query workload can get eventually extreme and it will be nearly impossible to efficiently do JOINSs at that scale.

Also schema flexibility (migration) is not trivial at large scale, but schema changes can be gradually introduced with NoSQL.

There are number of industries where companies did not even try to use RDBMS, they instead built databases from scratch. For example Facebook uses in memory (terabytes of RAM) graph database for connections between people.

1.3.1. Fixed Schema

In relational databases, we define schema (entities, attributes, indexes...) before we start using data.

But sometimes we must modify schema. Because of the big competition we must add/change features during development quickly, and this usually requires changes on the data model and corresponding parts of database schema. Adding, Modifying and Deleting index or column may lock rows or even table. Imagine this on large scale application with millions of rows.

1.4. Architecture

Over years application architecture has changed. Let's look at the last decades:

1980s: mainframe applications, one application with one database (example on Figure 4)

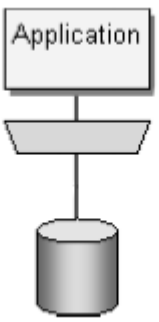


Figure 4 – mainframe architecture [4]

1990s: database as an integration hub (example on Figure 5)

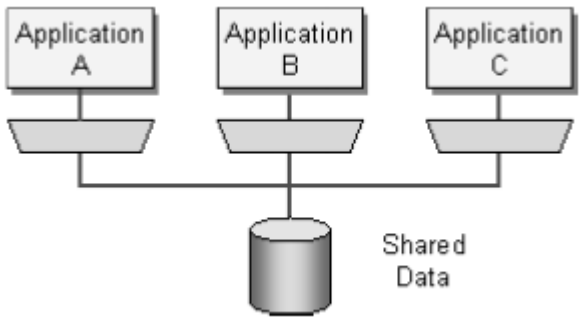


Figure 5 – integration hub [4]

We had a number of applications and they were connected to one database, and they shared data through that database.

2000s: decoupled services with own back-end (example on Figure 6 – image was inspired by Emil Eiframe's lecture about graph databases [4] but there were some mistakes so I corrected them)

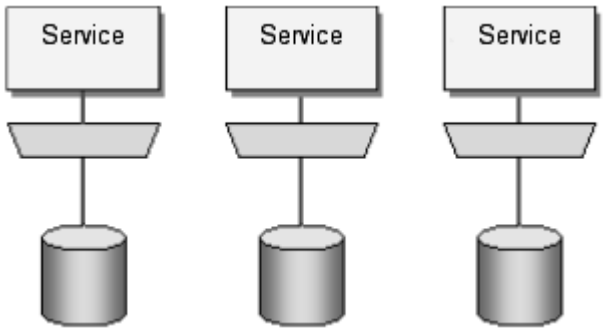


Figure 6 – decoupled services

Nowadays we build (especially web) our applications more as service oriented architecture; we compose our applications by number of services. So when we find database that is more suitable for one service, we can swap them and make our service faster.

And this is why (because of the services) we can start using NoSQL databases; organizational impact is today much less that it was in the past. We can use NoSQL as **purpose optimized storage** (e.g. searching, caching).

There are two current architectures that we should also mention.

1.4.1. Web 2.0

The term Web 2.0 refers to stage of evolution of World Wide Web where stationary content of web pages was replaced by shared space, interoperability and collaboration on content. It is closely associated with O'Reilly Media Web 2.0 conference in 2004 [5].

1.4.2. Software as a service

Software as a Service (SaaS) is a delivery model where software is hosted centrally and used by clients by small client or web browser. It was created as reaction to lowering software/server license costs and outsourcing.

2. SQL

2.1. Introduction

Although everybody is likely familiar with them, let's turn our attention to some fundamentals in relational databases. This will give us basis on which we can consider advantages and trade-offs of recent NoSQL data stores especially on large distributed data systems, such as those required at web scale.

There are many reasons why relational databases became so popular in last 4 decades. An important one is the Structured Query Language (SQL) which is feature rich and uses simple declarative syntax. SQL was adopted as ANSI standard in 1986 since that time it went through many revisions and also was extended by proprietary syntax such as Microsoft's T-SQL and Oracle's PL/SQL to provide additional features. SQL is easy to use, the basic syntax can be learned quickly, and junior developers can become proficient fast. Also there are many tools that include intuitive graphical interfaces for viewing and working with our database. In part because SQL is a standard, it allows us to integrate with various systems, and all we need is an application driver. If we decide to change our application implementation language or RDBMS we can do that often without bigger problems if we had not used too much proprietary extensions.

2.1.1. Transactions

In addition to features mentioned relational databases, also support transactions. As Jim Gray defines them: *A database transaction is a transformation of state with ACID properties* [6]. Key feature is that they execute at first virtually, allowing the programmer to do undo (ROLLBACK), and if everything goes fine transaction will be committed to database. Transactions are very important when we talk about comparison between SQL and NoSQL, so we must mention them more in deep.

ACID is acronym for Atomic, Consistent, Isolated and Durable - which are key features for transaction being executed properly.

- **Atomic** - every update within transaction must succeed in order to be successful, operation cannot be divided and all of them must be successful

- **Consistent**–means that data moves from one correct state to another correct state, e.g. we cannot delete user and leave his discussion post referencing his primary key untouched
- **Isolated**–two (or more) transaction cannot modify same data in same time, they must wait for another to complete
- **Durable**–if transaction succeeds changes cannot be lost

Problem with updates is that they take some time so transactions can become difficult under heavy load. When we attempt to horizontally scale our database (making it distributed), we now must think about distributed transactions. They will now spread over multiple systems. In order to keep ACID properties transaction, manager has to do some heavy work over multiple nodes (e.g. Two-Phase Commit) as Google architect Gregor Hohpe said in his post. [7]

2.1.2. Schema and Impedance mismatch

Often mentioned feature of relational databases is the rich schemas they afford and how we can represent our objects in a relational model. There is a whole industry of (expensive) tools for that. However, if we want to create properly normalized schema, we are forced to create tables that do not exist as objects in our application. For example, we have Students table and Courses table. But to represent many-to-many relationship we have to create sparse (sometimes also called JOIN) table to connect these two tables. This is slowly pollutes our data model where we prefer to have just courses and students. We must also do complex JOIN queries to connect these tables together. And that, as we mentioned in thesis introduction, can turn to be performance problem for our application.

This is called Impedance Mismatch. By definition impedance means that two objects lack some expected structural similarity. Key word here is expected, in ordinary circumstances we would expect that application will mirror into database. It is a result of the differences in structure between a normalized relational database and a typical object oriented class hierarchy. Databases do not map naturally to object models.

As Bryan Duxbury said in his post: *Impedance mismatch is a more subtle and challenging problem to get over. The problem occurs when more and more complex schemas are shoehorned into a tabular format. The traditional issue is mapping object graphs to tables and relationships and back again. One common case where this sort of problem comes to light is when your objects have a lot of possible fields but most objects don't have an instance of every field. In a traditional RDBMS, you have to have a separate column for each field and store NULLs. Essentially, you have to decide on a homogeneous set of fields for every object. Another problem is when your data is less structured than a standard RDBMS allows.* [8]If we will have an undefined, unpredictable set of fields for our objects, we will end with generic field schema with many JOINs (Object has many fields).

2.1.3. Scaling

As always easiest (and least disruptive) way how to scale our database is scaling up, but it has limitations and serious downsides mentioned in Chapter 1.This is the time, when we usually begin to think about scaling out. When we try to shard it seems natural that we need to find a key how to divide our data. E.g. when we have large customers table, likely is not a good strategy to divide rows on machines that every have only customers with same first letter of last name because machines with letters Q or X will sit idle while N, M or J will spike. We should shard by something random, numeric like a phone number or record creation. All in order to get our data better distributed.

There are 3 basic strategies for determining shard division:

- Feature-based shard – this approach chose eBay [9]in 2006 when they needed to support billion of queries a day. Using this strategy we split database by tables that are connected by same features and they not overlap too much. E.g. on eBay users are in one shard and items for sale are in another.
- Key-based sharding – we try to find key in our data to evenly distribute them across shards. Common strategy is use hash function on time based or numeric columns.
- Lookup table – we use one node in server cluster as "yellow pages" to the rest nodes. There are some obvious disadvantages: performance spikes on lookup

node every time we will access our data. This is called bottleneck because if lookup node is slow – everything gets slow. Also it is single point of failure.

2.1.4. Other

We preferably want to avoid going to disc as far as possible and serve out of main memory to get best performance and faster response time. Most relational systems are not memory oriented but disc oriented. Even with large main memory, RDBMS will end up going to disc for most queries. They are not aggressive about serving data from main memory and avoiding going to disk. Facebook tried to address this by building large in-memory MySQL cluster but they ended in using NoSQL solution anyway.

2.2. MySQL

MySQL is an open-source RDBMS that runs on a server and is a core part of widely used web application software stack LAMP (acronym for Linux, Apache, MySQL, and Perl/PHP/Python). It can run on most of Unix, Linux and Windows platforms. It has also a rich portfolio of management and development tools available.

In classical medium scale deployment MySQL can by Scaled Up by adding more powerful hardware and gigabytes of memory.

But on larger scale, we need to Scale Out on multiple servers to improve reliability and performance. MySQL uses simple one-way Master-Slave replication. All SQL statements are saved in binary log so they can be easily replicated on slave machines. Master is used for writes and his slaves are used to improve performance for reads as we can see on Figure 7 (this is typical high-end use e.g. on Facebook [10]). But when Master fails, we can set manually one of the slaves as new Master.

More performance can be added by caching databases queries in memory or by sharding. But with sharding we cannot use cross-shard SQL queries – if we do JOIN operation on data resided on single shard there will not be any problem, but over different shard we can get incomplete result set [11]. The best solution is to design application that will not need cross-shard queries – related data will reside on same

shards. Also cross shards queries and transactions can slow-down the whole database.

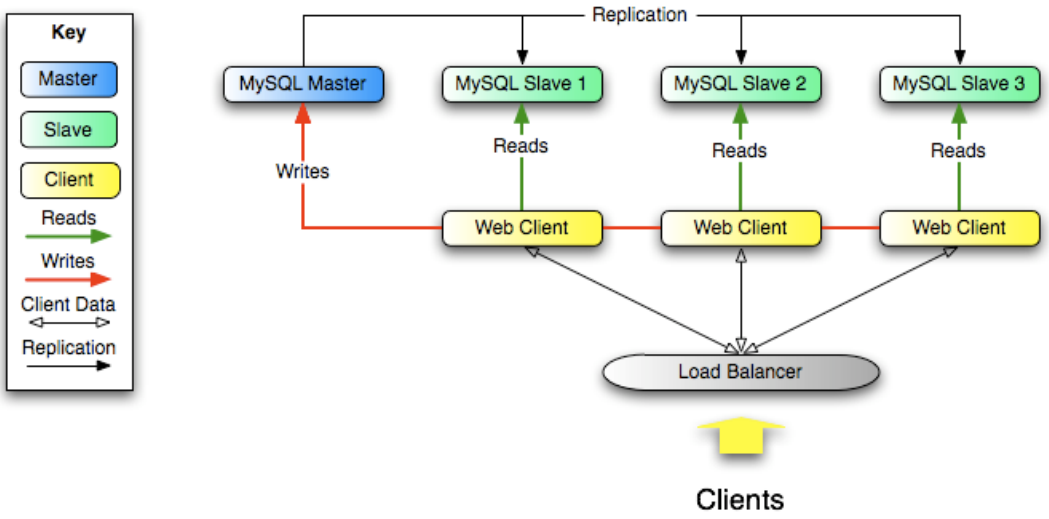


Figure 7 – Using MySQL replication for Scale-Out [12]

2.3. SQLite

SQLite is relatively small (about 300kB) ACID-compliant, embedded relational database. It was created in 2000 by Richard Hip as a small database for guided missile destroyer [13]. As an opposite of other relational databases it is integral part of client application, so there is no installation and no server. Also there is only one database file in cross platform format.

It implements most of the SQL-92 standard but it does not support RIGHT/FULL OUTER JOINS and only basics from ALTER TABLE.

Based on sqlite.org benchmarks [14] SQLite is very fast compared to MySQL and when we postpone writes to disk, it is even faster than MySQL in every test.

2.4. Microsoft SQL Server

SQL server is RDBMS developed by Microsoft and it uses proprietary SQL extension T-SQL. Server can run only on Windows systems but has very rich bundle of management and development tools from Microsoft available.

Replication services supports 3 types of replication:

Transaction replication – every write transaction made to master database is synced out to slaves in near real time

Merge replication – write changes are made to both master and slaves and synchronized bi-directionally, conflicts are solved by predefined policies. It is more close to Master-Master replication.

Snapshot replication – master database creates its snapshot and then replicate it to Slaves, changes are not tracked.

Sharding support with SQL Server is not included in his RDBMS so we will face same problem I mentioned in this chapters Introduction.

2.5. Object Relational Mapping

ORM is technique of converting data between incompatible type systems (in our case RDBMS) and object model used in object-oriented programming languages. This creates "virtual object database" effect in programming language. We usually we have a Persistent layer which does every work for us connected with saving, modification, reading and deleting data.

One of the most famous implementation is Hibernate library for Java or his derivate for .NET NHibernate. Both can map to most of RDBMSs. We can see example on Figure 8.

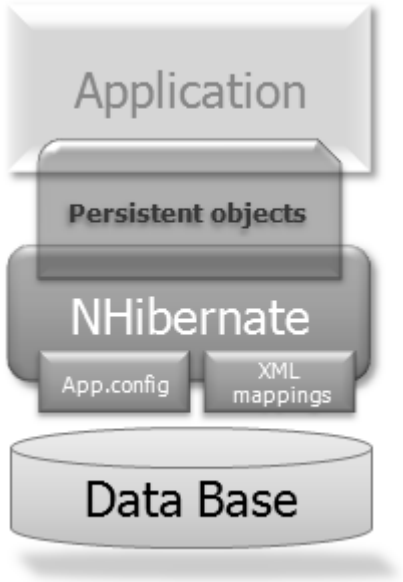


Figure 8 – NHibernate architecture [15]

Biggest advantage of ORM is fast development or easy database migration. If we will use specialized API for used RDBMS we must usually rewrite some parts of app for database migration which will cost us time and money. With ORM we can usually just switch to another database.

Now disadvantages. We will get big performance drop [16] and by nature of ORM sometimes it is not even possible to get better performance (again see results in [16]). The majority of ORM users are Java and .NET developers. Sometimes implementation of this technology leads to poor code or bad application performance because programmers are not forced to think about database.

ORM has today definitely place in application development because sometimes fast development is often the biggest concern in private companies. But also when it comes to performance, it is better to look also at NoSQL.

2.6. Conclusion

Relational databases are very good at solving certain data storage issues, but because of their background, they also can create problems of their own when it is time to scale. When we try to minimalize number of JOINs we must denormalize our schema and that means maintaining multiple copies of our data and seriously disrupting our design by “plumbing” code.

Further, you almost certainly need to find a way around distributed transactions, which will quickly become a bottleneck. These compensatory actions are not directly supported in any but the most expensive RDBMS. And even if you can write such a huge check, you still need to carefully choose partitioning keys to the point where you can never entirely ignore the limitation. [9]

In the end, when we see limitations of RDBMS and strategies developers use to overcome their scaling problems NoSQL solutions are maybe not that radical for us and maybe more natural with design and managing large amounts of data.

3. NoSQL

3.1. Introduction

NoSQL movement began in early 2009 on "NOSQL meetup" organized by Last.fm to discuss open-source distributed databases [17]. Term NoSQL is usually [18] defined as data store addressing most of these points: non-relational, distributed, horizontally scalable and often schema-free, easy replication support, eventually consistent (we will explain these terms later in this chapter). So we can best describe NoSQL as an abbreviation of "Not Only SQL".

3.1.1. Brewer's CAP theorem

In order to understand NoSQL databases we need to understand CAP theorem. It was introduced by Eric Brewer at the 2000 Symposium on Principles of Distributed Computing [19] and formally proved in 2002 by Seth Gilbert and Nancy Lynch [20].

The theorem states that for a distributed data system it's impossible to provide all of these three requirements: Consistency, Availability and Partition Tolerance. According to theorem, we can satisfy only two of the three because of their mutual sliding dependency (Figure 9 illustrates visually CAP theorem as Venn diagram). E.g. more consistency we demand from system, the less partitioning we can do unless we tune down availability of system (for example by locking).

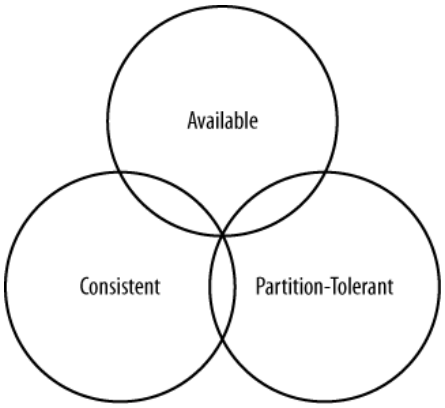


Figure 9 – Venn diagram of CAP theorem [9]

Let's look more closely on CAP requirements.

Consistency

Consistency means that all database clients will read the same value for the same query, even when it was given by concurrent updates.

There are three consistency models available with NoSQL databases:

Strict (sequential) consistency

Strict consistency is strongest level of consistency meaning every read will return most recently written value. On single machine this does not pose any problem, but on global scale with distributed data centers it requires e.g. some sort of global clock timestamping every operation or maybe some global lock…but that can seriously slower whole distributed data store and create bottle neck.

As Amazon CTO Werner Vogels puts it: *Data replication algorithms used in commercial systems traditionally perform synchronous replica coordination in order to provide a strongly consistent data access interface. To achieve this level of consistency, these algorithms are forced to tradeoff the availability of the data under certain failure scenarios. For instance, rather than dealing with the uncertainty of the correctness of an answer, the data is made unavailable until it is absolutely certain that it is correct* [21].

Causal consistency

Causal consistency is weaker form of strict. This model attempts to determine the cause of events to create some consistency from their order. So when we read data, we must read them in sequence if there were written as potentially related.

Eventual (weak) consistency

This model means that every update will propagate through all replicas in distributed system and eventually all replicas will be consistent.

This is actually not that big problem as it could look like. Most systems will need more availability and partition tolerance than strong consistency. E.g. as former

Microsoft India Web Platform Lead Vineet Gupta said in his post: C*ustomer wants to place an order – you will accept the order, not return the money saying the system is unavailable – availability is important. Inventory would be checked asynchronously. Order details would be checked asynchronously. All this while data would be in an inconsistent state* [22].

Availability

Data store must always allow clients not only to read data but also to write them.

Partition Tolerance

The data store can be partitioned on different machines, and it will continue working when some messages will be lost or even some machines fails.

But when we talk about distributed systems and network partitioning, we must see that in some level machines fails continuously and packet loss is inevitable. Because of that distributed system must be Partition-Tolerant. So that leaves us only two options to choose Consistency and Availability.

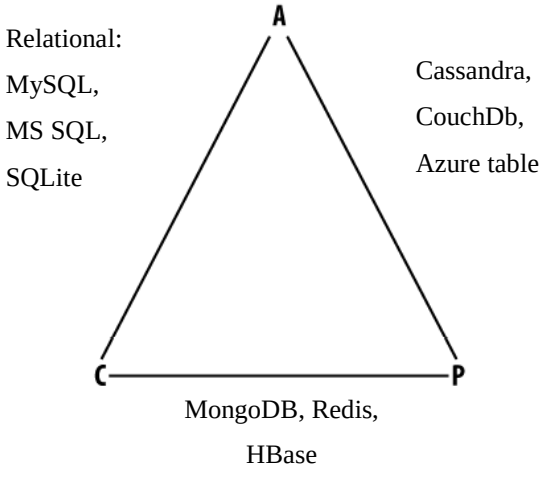


Figure 10 – where different databases appear in CAP taxonomy

Figure 10 shows where different databases from next chapters are placed; it was inspired by CAP image from Eben Hewitt's book Cassandra: The Definite Guide [9]. However I added some data stores based on my research. Note that placement can change by configuration of this data stores.

3.1.2. MapReduce

If background batch processing is our problem and we are not aware of MapReduce model, we should be. MapReduce is a software framework developed (and inspired by functions Map and Reduce from functional programming) by Google to work with large data sets on computer clusters. It is a way of writing batch processing jobs without having to worry about infrastructure.

MapReduce works in two phases:

Map: master node accepts a request and divides it to subrequests which distributes to worker nodes. Worker nodes will execute subrequests and return answer to the parent node.

Reduce: when a master has enough answers or time limit for answering expires, it will execute Reduce function. It combines/reduces data in a way defined by user. Then it returns result to the user.

Biggest advantage is that Map can be performed in parallel and send their data to "reducers". In the result MapReduce can be applied to very large datasets, e.g. large server farm can using MapReduce sort petabyte of data in only a few hours [23].

Different databases work more or less fluently with MapReduce concept - keep that in mind when we choose database to fit our needs. Hadoop (see 3.4.3) is one of the biggest open MapReduce implementations and MongoDB (3.3.3) also includes some MapReduce ideas on a smaller scale.

Let’s show MapReduce on MongoDB example, imagine we have rows from text document saved as strings in database. We want to compute word frequency. In Figure 11, we can see example of Map function. Database will run Map function on every row of the document and Map function will split the row into an array with the word as key and count=1 as value. Now database will group key value pairs by their key and run Reduce function from Figure 12 on them. Reduce function will count the sum of all counts. Then database will group results again and run Reduce function on them. This will loop until combination of group and Reduce function stops producing new results as we can see on Figure 13.

```
function wordMap(){

      //find words in the document text
      var words = this.text.match(/\w+/g);

      if (words == null){
            return;
      }


      for (var i = 0; i < words.length; i++){
            //emit every word, with count of one
            emit(words[i], {count: 1});
      }
}
```

Figure 11 – MongoDB map function example [24]

```
function wordReduce(key,values){

      var total = 0;
      for (var i = 0; i < values.length; i++){
            total += values[i].count;
      }
      return {count: total};
}
```

Figure 12 – MongoDB reduce function example [24]

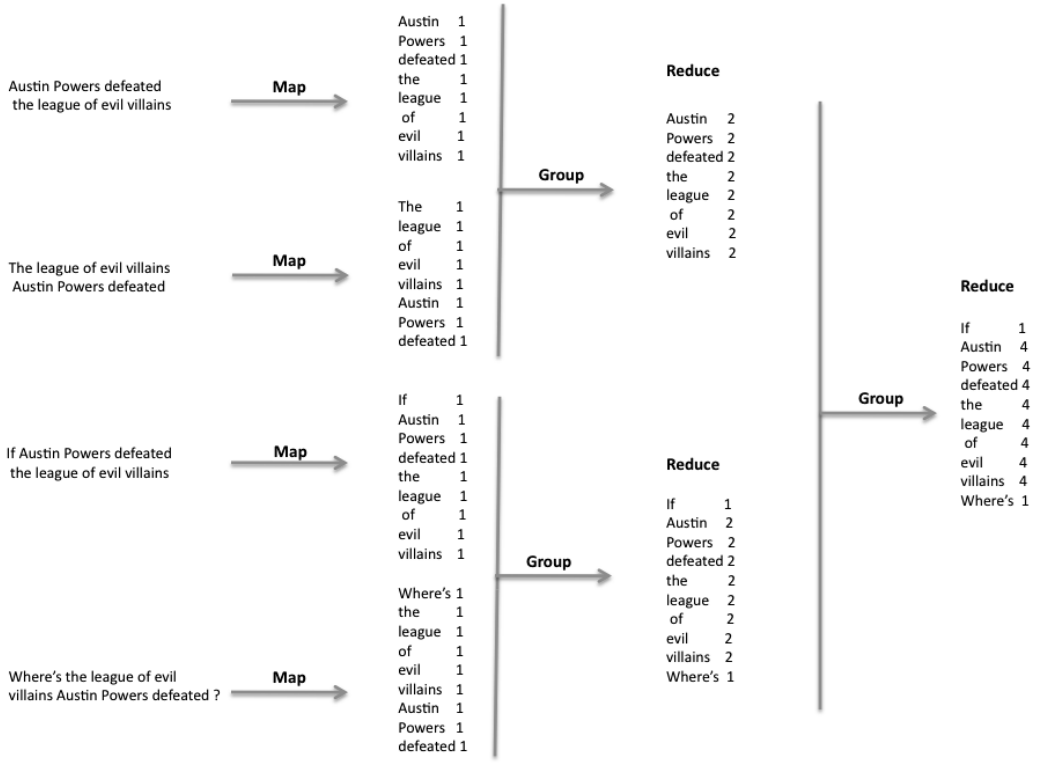


Figure 13 – MapReduce example [24]

3.1.3. Notes on taxonomy

Now, we have in my opinion some foundations to look on concrete NoSQL data stores. I chose taxonomy by data storage mechanism because I think it is more solid and more widely used than taxonomy by CAP theorem.

3.2. Key-Value Stores

3.2.1. Introduction

They provide the simplest possible data model. Usually it is **collection of (distributed) key-value pairs (hash tables)**.We can retrieve item based on its key, we can insert key/value pair and we can delete a key/value pair. They are focused on scaling to huge amount of data and designed for massive load. However that comes with the cost. Range queries are not straightforward (unless the database provides explicit support) and if we use only key value stores for our application it can complicate development. Usually they are schema-less and version data with datastamps. Majority of K-V stores are based on Amazon's Dynamo paper [21].

So in short: an extremely simple data model (example on Figure 14), which means scaling out is easy but also means it is poor in handling complex data.

Car	
Key	Attributes
1	Make: Nissan Model: Pathfinder Color: Green Year: 2003
2	Make: Nissan Model: Pathfinder Color: Blue Color: Green Year:2005 Transmission: Auto

Figure 14 – example of typical Key-Value pairs [25]

3.2.2. Azure Table Storage Services

One of the services provided by cloud platform Windows Azure is Table Storage. By Microsoft's definition Table Storage is queryable structured storage for non-relational data. Table Storage Services does not have a fixed schema; entities in table can have different structure and different properties. Scheme can change on client side.

Data Model

Each Azure Storage account has collection of unlimited number of tables with no limit on table size. Tables are collections of entities (similar to rows in RDBMS).We can look on illustration in Figure 15. Each entity has three properties, the PartitionKey, the RowKey and Timestamp that are not shown in above for space/legibility reasons. Together these form a unique key for an entity. An entity also has a set of properties (Columns). A property is a name-value pair, same as a column. Additionally, currently the only index and all results are returned sorted by PartitionKey and then by RowKey.

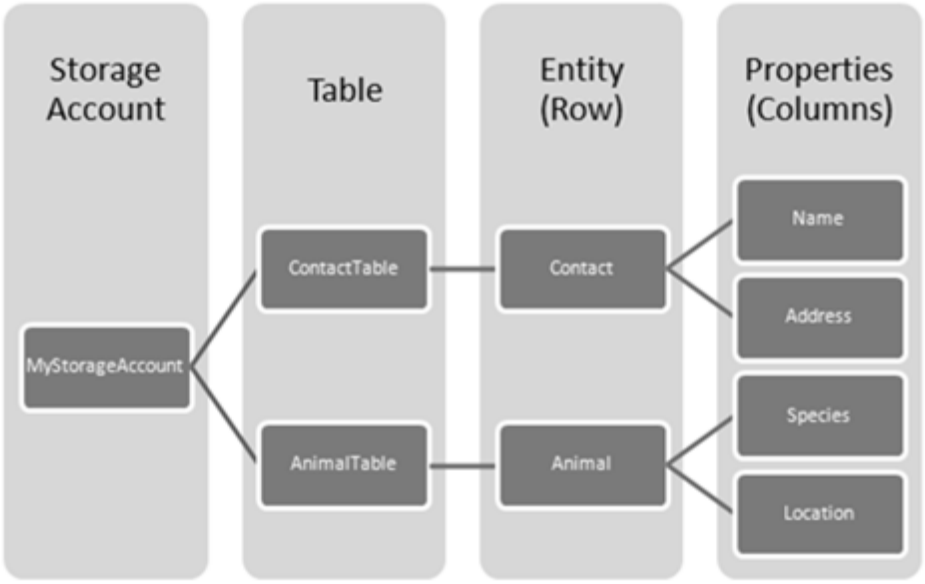


Figure 15 – Azure Table Storage data model [26]

In short data model are collections of free form entities with 3 mandatory properties:

- **PartitionKey** - tables are partitioned in nodes to support load balancing, partition has entities with the same PartitionKey value
- **RowKey** - is unique in a partition
- plus also Timestamp - read only, set by server

Remarks

To access Azure Storage we can use REST and we will receive data in ATOM/XML format. If we use application in Azure cloud we can also use LINQ. Application can access Storage no matter if it is hosted on Windows Azure cloud or in-premise server.

Main problem with Azure Table Storage is that it is quite young technology and has some serious limitations in querying and also in data size: Maximal query size is 1000 entities and maximal execution time is 5s. Transaction limitation is 100 operations and 4MB of data. Also currently does not support server side procedures.

3.2.3. Redis

Redis is basically in-memory database, it keeps the whole dataset in memory and sync to disc after a defined time (it uses append-only file). It is very small (~16k lines of C) and very fast - there is no notable difference between read and write operations. So it is very quick and fast storage. Also have bindings to most of common languages (C/C++/C#, Java, PHP…) [27]. Development is now funded by VMware.

Values in Redis can be Strings as in a conventional key-value store, but also Lists, Sets, and Sorted Sets (to be support in version 1.1). This data types allow pushing/poping elements, or adding/removing them, also perform server side union, intersection, difference between sets, and so forth depending on the types. Redis supports different kind of sorting abilities for Sets and Lists.

You can think in Redis as a Data Structures Server, that allows you to model non trivial problems. Read Data Types to learn more about the way Redis handle Strings, and the Commands supported by Lists, Sets and Sorted Sets. [28]

Redis supports master-slave replication. Data can replicate to any number of slaves - that means read (not write) scalability. Write scalability must be implemented in application.

3.3. Document Stores

3.3.1. Introduction

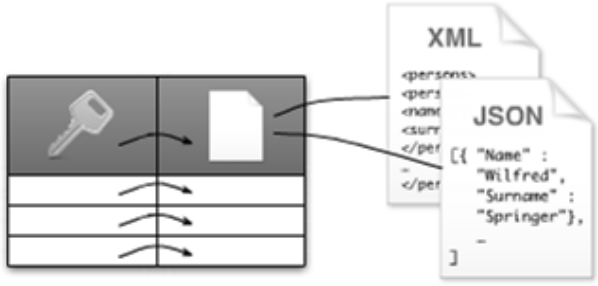


Figure 16 – simplified document store data model [29]

Document oriented databases stores information as documents of related data. All of the data in a document is self-contained, and does not rely on data in other

documents within the database. A document in a document database is typically a tree of objects containing attribute values and lists, often with mapping to JSON[3] or XML often versioned, or we can say that data-model is a collection of key-value collections (on Figure 16 we can see very simplified example). They have flexible schema, similar to key-value stores but value is a document. Document stores has (compared to key-value stores) improved indexing and server side processing.

For some applications data integrity is not a primary concern, such databases can work fine without restrictions provided by relational databases, which are designed to preserve data integrity. Instead giving up on these restrictions gives up document databases possibility to provide functionality that is difficult or maybe even impossible to provide with a relational databases. So it is trivial to set up a cluster of document oriented databases, making it easier to deal with some certain scalability and fault tolerance issues. Such clusters can theoretically provide us with limitless disc space and processing power. This is primary reason why document databases (and key value stores) are becoming the standard for data storage in the cloud.

Is this different from just dumping JSON strings into MySQL? Document databases can actually work with the structure of the documents, for example extracting, indexing and filtering based on attribute values within the documents. Alternatively we could of course build the attribute indexing ourselves.

Big limitation of Document Stores is that most implementations cannot perform joins or transactions spanning several documents. This restriction is deliberate because it allows the database to do automatic partitioning which can be important for scaling. If the structure of our data is lots of independent documents, this is not a problem - but if our data fits nicely into relational model and we don’t need joins, please don’t try to force it into document model.

---

[3]JSON, which means JavaScript Object Notation, is a data exchange format developed as an alternative to XML. It supports only a few data types: number, Unicode string, boolean, array, object, and null.

3.3.2. CouchDB

CouchDB is document oriented database stores data as JSON documents without fixed-schema. Also has powerful views which query the database and computes calculations on documents. Uses distribution of data by replication, uses bi-directional replication (Master-Master) with bi-directional conflict detection/resolution, can run offline, and then sync back changes. Also does not have JOINs between documents, primary keys or foreign keys (UUIDs are automatically assigned and stored in B-Tree Storage Engine).

Development began at 2005 by former Lotus Notes Developer Damien Katz. Couch means "Cluster Of Unreliable Commodity Hardware". Now Apache Top Level Project (Licensed under Apache License) commercially supported by CouchDB. Written in Erlang - functional, concurrent oriented programming language (created by Ericsson for telecommunication).

View engine

View engine is something that should catch our interest. It uses MapReduce "views" dynamically generated by JavaScript to do sort of bridge between NoSQL and relational databases. These views map the document data onto a table-like structure that can be indexed and queried. Views can be rebuilt whenever it is necessary or can be configured to return stale data. Couch DB views use MapReduce approach to selecting documents from database. Reduce function is optional. JavaScript is default language for MapReduce functions. Because of its View engine CouchDB is ideal as "archive" database with Views returning stale data.

Documents

Couch DB documents are very flexible. JSON format (example on Figure 17) is allowing us to take advantage of JSON array and dictionaries to represent collections of data. There is nothing to dictate how a document should be structured, or what it should contain (as long as it is valid JSON format).

```
{
   "_id": "CouchDB: Databases and Documents",
   "_rev": "1-704787893",
```

```
   "author": "John Wood",
   "email": "john_p_wood",
   "post":  "CouchDB  is  a  documented  oriented  database.  A
document...",
   "tags": ["couchdb", "couchdb case study", "json"],
   "comments": [
      {
         "email": "joe@somewhere.com",
         "comment": "Thanks for the information"
      },
      {
         "email": "kevin@xyz.com",
         "comment": "CouchDB sounds pretty interesting"
      }
   ]
}
```

Figure 17 – Document example for blog post [29]

3.3.3. MongoDB

Development started in 2007, commercially supported and developed by 10Gen. The MongoDB team aims to be "MySQL of NoSQL" they tries to be truly universal NoSQL data store. On Figure 18 we can see where they assume their position in data stores world.

Figure 18 – MongoDB position [30]

MongoDB is open source non-relational document database (example of document on Figure 21) that combines three things: scalable, schema-less and queryable. It has native drivers for almost every major language. Mongo does not implement a few

features of RDBMS (e.g. joins or transactions) in order to achieve much better performance and horizontal scalability. Compared to CouchDB, Mongo has much better querying capabilities (example on Figure 20): we can use dynamic object-based queries (also against embedded objects and arrays) without pregenerating expensive views.

Atomic transactions over multiple documents or collections are not allowed in MongoDB. But we can do atomic transactions (e.g. update comment posts in blog) on a single document, including all of its embedded objects. Part of the reason why MongoDB so fast is because it not requires locking. In situation when we update post and comments using RDBMS we will need lock multiple tables, but using MongoDB these will be included in single document.

Scaling out via sharding

Mongo has automatic sharding (scaling out) feature to distribute data and load across multiple servers. It uses consistent hashing (like Amazon Dynamo), order preserving and range chunks for splitting data into shards. This idea comes from Google BigTable (they call it tablets). They get ASCII name key to create ranges and by that range data spreads on shards. Chunks also help with range queries (example on Figure 19).

collection	minkey	maxkey	location
users	{ name : 'Miller' }	{ name : 'Nessman' }	$shard_2$
users	{ name : 'Nessman' }	{ name : 'Ogden' }	$shard_4$
...			

Figure 19 – Example of MongoDB shards [30]

```
// select * from posts where 'economy'in tags
// order by ts DESC limit 10

//Support multiples types of indexing
db.posts.ensureIndex({tags:1});
db.posts.ensureIndex({ts:1});
db.posts.ensureIndex({tags:1,ts:-1}); // compound

cursor =
db.posts.find({tags : 'economy'}).sort({ts:-1}).limit(10);
```

Figure 20 – MongoDB Query Example [30]

```
{
title: ‘Too Big to Fail’,
text: ‘article text here...’,
author: ‘John S’,
ts: Date(“05-Nov-09 10:33”),
comments: [ { author: 'Ian White',
comment: 'Great article!' },
{ author: 'Joe Smith',
comment: 'But how fast is it?',
replies: [ {author: 'Jane Smith',
comment: 'scalable?' } ]
         }
             ]
   ,
tags: [‘finance’, ‘economy’]
}
```

Figure 21 – Blog Post Data Model Example [30]

Note for Figure 21: comments are included in object because we cannot use joins in MongoDB.

3.4. Column Oriented Stores

3.4.1. Introduction

Column oriented stores are inspired by Google's BigTable paper [31]. The BigTable paper describes development of distributed and scalable database BigTable for Google services. Sometimes they are also called hybrid row/column stores. They are like a column oriented Relational DB, but with a twist, the data model differs from RDBMS: Unique thing about BigTable is that every row has an individual schema (it is not pre-defined) and can have different set of columns. Empty columns in rows are not stored at all. This can mean huge savings in both disk space and IO read time. We can set that this row will have 24 columns, but this row only 3 columns. This also allows us to essentially store one-to-many relationships in a single row, if our child entities are truly subordinate, they can be stored with their parent, eliminating all join operations.

BigTable’s data model was inspiration for many projects. The most popular are HBase, Hypertable and Cassandra. Because they does not have predefined schema, they are very attractive for applications where we do not know in advance the attributes of objects or they change frequently. Also these databases are very good for high distribution.

Column Oriented Stores shares same limitations with Document Stores. We cannot perform joins or transactions spanning several documents because of automatic partitioning.

3.4.2. Cassandra

Cassandra was prophetess in Troy during the Trojan War. Her predictions were always true, but never believed. Some sources call it first 2nd generation NoSQL database, because it combines the data model of 1st generation NoSQL data stores BigTable with lot of scale out distributed characteristic of Dynamo. Created at Facebook for Inbox search and now was released open-source to Apache Software Foundation.

Because of her Inbox origin, Cassandra is internally optimized towards heavy-write systems and designed to be high available and eventually consistent on distributed systems. Let's look on that more thoroughly.

Data model

The Cassandra data model is inspired by BigTable model but has a lot of different features.

Column–named column is basic unit of storage in Cassandra, it consists of Name-Value pair and timestamp for write conflict resolutions on server side (last write wins).

Column Family - Collection of similar data, indexed by Row Key. For example we can have User column family, SchoolSubjects column family… It is a container of rows that have columns sets, we can picture it as analogue to tables in RDBMS – BUT column sets not needs to be similar (as we can see on Figure 22).

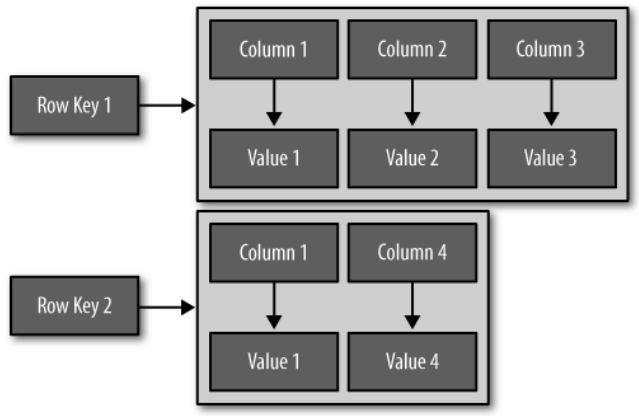


Figure 22 – example of a column family [9]

Super Column - Columns who holds collection of columns, when we need to create a group of related columns. As we can see on Figure 23 it can be placed in column family as regular column.

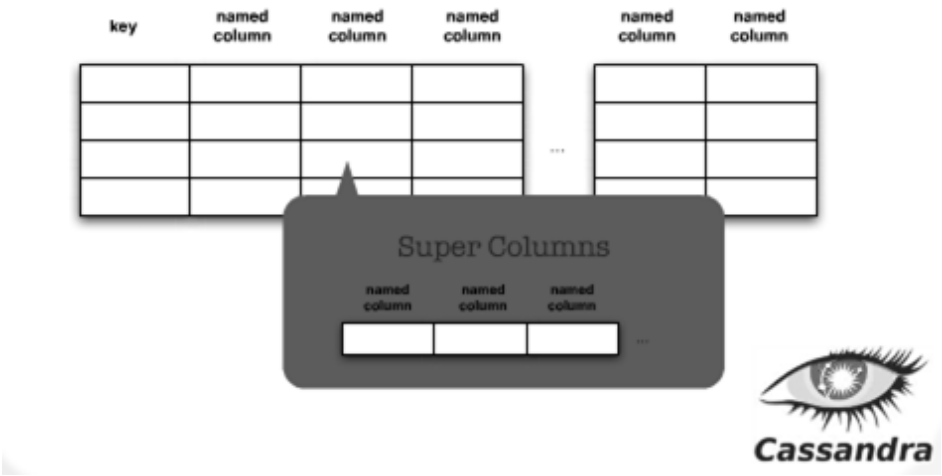


Figure 23 – example of super column [29]

Availability

Cassandra uses Master-Master replication, co it is decentralized and distributed – we can have single cluster running across possibly hundreds geographically dispersed data centers. At this scale, small and large components fail continuously, so Cassandra is designed for that by flexible replica replacement – we can replace failed nodes in cluster with no down time. Cassandra is designed for high availability (reads will always succeed).

Scaling

Cassandra uses special property of horizontal scalability called elastic scalability. It means that cluster can seamlessly scale up and down by needs of user without major disruption or reconfiguration of entire cluster. Data will automatically rebalance.

Cassandra uses three strategies for partitioning:

- Random - good for distribution of data between nodes, but disables range queries
- Order Preserving - can lead to unbalanced nodes, but allows range queries
- Custom

Consistency

Consistency essentially means that a read always returns the most recently written value. Cassandra trades-off some consistency for achieving total availability. We can tune that between eventual consistency (Cassandra is internally optimized for this model) and strong consistency (not recommended by Cassandra developers because of performance) by setting consistency level. It is called **tuneable consistency**. Consistency level sets the number of replicas that must reply that write was successful. So we can say that on consistency settings recommended as best practice all reads will always succeed but may not all return same data. At Figure 24 we can see example of Cassandra's replication on 3 nodes.

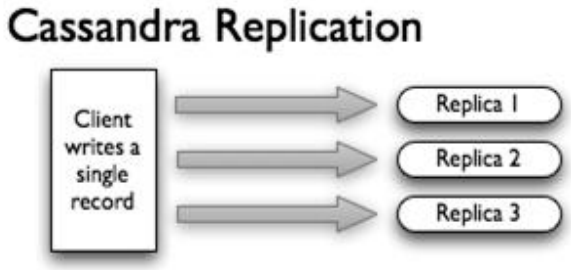


Figure 24 – Illustration of Cassandra's replication [32]

3.4.3. HBase

HBase is the Hadoop database. Hadoop is a set of open source projects that deal with large amounts of data in a distributed way. It contains Hadoop distributed file system (HDFS) and also MapReduce subprojects are open source implementations of Google's GFS and MapReduce.

It is based on Google's BigTable; we can say it is BigTable written in Java. The project goal is hosting very large tables (billions of rows with millions of columns) on clusters of commodity hardware.

Main features are: highly distributed, data is stored sorted (no secondary indexes), automatic partitioning, re-balancing, and re-partitioning.

Data model

Data model of HBase is very similar to traditional BigTable and Cassandra, so I will only mention some interesting things.

HBase columns are called cells. HBase is very good for versioned data; we can use cell's timestamp for that. E.g. when we want to save users locations, we can just add them into column family with different timestamp. Now we have user's location history in one place and, again, no joins needed.

Consistency& Distribution

HBase's replication model (actually, it is the HDFS replication model) has very important feature called replication pipelining.

When a client is writing data to an HDFS file, its data is first written to a local file. Now as HDFS documentation says: *Suppose the HDFS file has a replication factor of three. When the local file accumulates a full block of user data, the client retrieves a list of DataNodes from the NameNode. This list contains the DataNodes that will host a replica of that block. The client then flushes the data block to the first DataNode. The first DataNode starts receiving the data in small portions (4 KB) writes each portion to its local repository and transfers that portion to the second*

DataNode in the list. The second DataNode, in turn starts receiving each portion of the data block, writes that portion to its repository and then flushes that portion to the third DataNode. Finally, the third DataNode writes the data to its local repository. Thus, a DataNode can be receiving data from the previous one in the pipeline and at the same time forwarding data to the next one in the pipeline. Thus, the data is pipelined from one DataNode to the next [33].We can see replication pipelining on Figure 25.

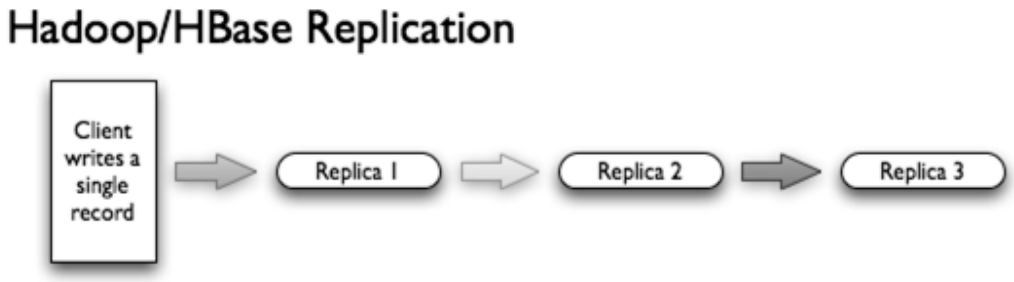


Figure 25 – Illustration of HBase DataNode replication [34]

Now we can see how replication pipelining guarantees data consistency, when write is completed every node will have same data and we can guarantee write order. This also helps some automatic fault tolerance - when during pipelining one node fails, HDFS automatically redirect query to another node. Also we can add new server and data will be easily replicated on him.

This may be main reason why Facebook chosen HBase over Cassandra for their messaging system [34]. HBase trades little of availability for achieving strong consistency.

3.5. Conclusion

As we saw, none of the so-called "NoSQL" databases have the same implementation, goals, features, advantages, and disadvantages. To help us to orient in this world I made Table 1:

Table 1 – Comparison of NoSQL storages

	Orientation	**Written in**	**Consistency**	**Distributed**	**Replication**
Azure	Key-Value	(proprietary)	Eventual	Yes	M-M
Redis	Key-Value	C	Strong	Yes	M-S
CouchDB	Document	Erlang	Eventual	Yes	M-M
MongoDB	Document	C++	Strong	Yes	M-S
Cassandra	Column o.	Java	Eventual	Yes	M-M
HBase	Column o.	Java	Strong	Yes	HDFS

More information about performance of NoSQL solutions can be found in experimental part section (4.7).

Now let's summarize main advantages and disadvantages of NoSQL.

3.5.1. NoSQL: The Good

Today's large (web) applications have some specific and also new needs, this is the list in which NoSQL databases excels.

Simple queries – most of the queries on web applications are by the primary keys, NoSQL databases usually work them really fast

Easy to use – lot of NoSQL solutions has today easy to use libraries, which makes them good choice even on one-machine use cases

Low Latency – because NoSQL transactions mostly does not require locking writes are very fast, also simple reads

Scalability, elasticity, geographic distribution – because of their simple data model they scale easier than RDBMS, elastically can scale by application needs (up and down)

High availability – because they trade some consistency (e.g. eventual consistency) for availability, or because of their implementation (e.g. HBase)

Flexible-schemas – because of their data model they easily handle semi-structured data

Smaller Impedance Mismatch– many NoSQL databases retain data in structures that map more directly to object classes used in the application code, this can significantly reduce development time

But we should also consider the down sides.

3.5.2. NoSQL: The Bad

Limited query capabilities (so far) – e.g. because of distributed nature of most NoSQL databases

Rough tools – because NoSQL databases are so young there are still not so much tools for them

Eventual consistency – can make client applications more difficult because of read conflicts

No standardization – can make problem with portability or create vendor lock-in[4]

Not bug friendly – integrity control of data often lies on application, RDBMS does that usually for us

[4]Vendor lock-in – economic term, makes customer dependent on one product or service, unable to switch to another without substantial costs

4. Experimental part

This chapter I will try to demonstrate differences of SQL and NoSQL solutions by a benchmark around a simple e-shop use case. I was very unsatisfied by existing NoSQL benchmarks because they mostly didn't reflected reality or wasn't using exactly best practices for tested databases. But then I discovered Michael Kennedy's post about MongoDB vs. SQL Server comparison [35]. He uses main application that runs small simple console clients and that little ones connect to databases and runs queries concurrently (start is synchronized by mutex). Other benchmarks were using single clients to run queries and that is, from my experience as web application developer, highly unrealistic scenario these days. But M. Kennedy in my opinion made some mistakes in his benchmark (e.g. bad data model for SQL Server) and wasn't using best practices. So I recreated some of his tests and added some of my benchmark own scenarios and technologies to test. I also added more easy to use graphic interface.

4.1. Scenario

Develop an application to support concurrent users. I will simulate that by creating multiple instances of small clients that will perform benchmark queries.

These are the main goals for benchmark:

1. Define classes from program perspective and database independent; sub-object will be collections in main objects.
2. Create "DataContext" [5] class for persisting objects in data store; every type of object will have a collection in DataContext class.
3. Develop a benchmark application

I chose these data stores:

[5] DataContext is a term commonly used in ORM world to describe source of all entities mapped from database

- Microsoft SQL Server 2008R2 as database server and Entity Framework: Code First for ORM. In my opinion ORM are so widely used in applications these days that I should use it also in this benchmark. Code First is also very easy-to-use ORM because very small amount of code is needed [36]. Also is in my opinion quite fast.
- Redis was chosen as representation of "brute-force" approach. In memory database should be very fast by its nature.
- At last I chose MongoDB for representing document stores because is advertised as very versatile data store.
- I didn't tested Cassandra and HBase because I have only one computer available and I think that without demonstrating theirs high distributivity this test will be useless. Azure storage also wasn't tested because of my low budget.
- I also didn't tested CouchDB because I personally think that MongoDB will perform better, and I also wasn't satisfied with its object-mapping tools.

4.2. Benchmark Use Cases and Datasets

4.2.1. Basic Inserts

Users will insert small independent pieces of data (class Basic) into database through 5 separated clients (e.g. like instant messaging or twitter).

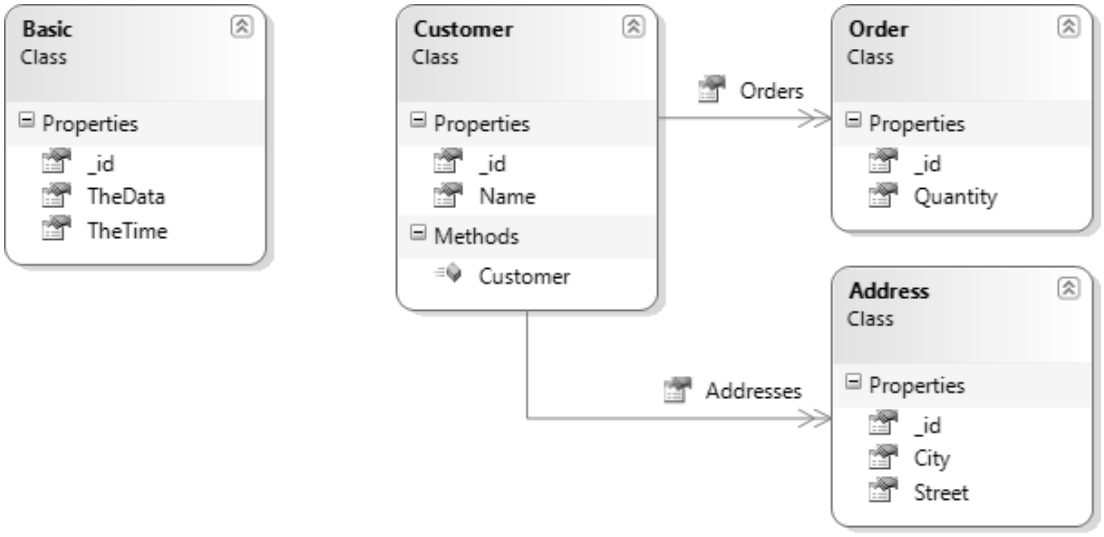


Figure 26 – Benchmark classes diagram (MongoDB version)

Class Basic is triplet of int (id), string (200 characters), and time of insertion as we can see on Figure 26.

4.2.2. Basic Search Queries on Primary Key

Users will repeatedly try to find objects by defined id through five clients, half of the searches will end in failure because id will not exist. Benchmark app will insert needed amount of data and perform defined search queries.

4.2.3. Basic Search Queries with Update

Users will repeatedly try to find objects by defined id through five clients and update its DateTime field. Benchmark app will insert needed amount of data and perform defined search and update queries. Half of the searches will end in failure because id will not exist. I think that we will see performance drop compared to previous use case on larger datasets.

4.2.4. Basic Search Queries on Non-Indexed field

Users will repeatedly try to find objects by defined string through five clients. Benchmark app will insert needed amount of data and perform defined search queries.

4.2.5. Complex Queries

In my experience most of the joins operations with objects are associated with object and his sub-objects. In classic RDBMS solution we usually use separate tables for each object and sub-objects and join them by queries, but NoSQL solutions allows you to include sub-objects inside main object so no additional operations are needed.

Because of reasons I mentioned in last paragraph I chose to measure performance on object Customer with 2 types of sub-objects: Address and Orders. These are representing list of customers which everyone has a lists of available addresses and list of his orders as we can see on Figure 26.

Benchmark will insert needed amount of data and then will query every main object through 5 clients including both sets of his sub-objects. Also it will perform sum of his all orders. I suppose that this test will show the cost of JOIN queries very clearly.

4.2.6. Dataset Scaling

Because every test is done by 5 clients, I set the dataset scaling at: 50, 100, 500, 1000, 5000, 10000, 50000 queries so every client will run one fifth of whole amount of queries.

4.3. Developer Documentation

4.3.1. Coordinating application

Coordinating application is a classic WinForms application for running benchmark clients and coordinating start of queries execution. Application will create mutex[6] of defined name and clients will pause on mutex after their initialisation. When we hit the "start benchmark" button in application all client will start executing queries.

4.3.2. Benchmark client basics

Every client das 3 numerical parameters first is use case number, second amount of inserts to be performed and last amount of queries to be performed.

After start every client will connect to database and perform defined initialisation for benchmark, and then it will wait for Coordinating App's mutex.

4.3.3. SQL Server Client

I personally followed development of Entity Framework: Code First very closely and I used in in many projects so I chose is as representing ORM for benchmark. Entity Framework is Microsoft's data access library for SQL databases. Code First is a

[6] mutex stands for „mutual exlusion", in .NET its a synchronization primitive, it can be used also for synchronization between processes [40]

development pattern that enables us to define C# classes first and these classes will be used to generate a database schema and will be mapped on them. Also supports lazy loading of sub-objects of objects by loading them on demand and by that minimising needs for excess JOINs. It's very good for fast development but still lacks good schema evolution tools. Also it support's LINQ[7] language for querying data.

Data model is quite ordinary, Entity Framework requires that primary key is named "Id" everything else is done by ORM.

4.3.4. MongoDB Client

We will need some strongly-typed interface for connection to MongoDB and very fast JSON to .NET objects de/serialisation. I chose NoRM library [37]. It uses LINQ query language so it's very easy to use to .NET developers.

In data model primary key should be type of NoRM.ObjectId (12-byte unique value) [38] and named "_id". Everything is same.

4.3.5. Redis Client

If we want to implement DataContext like class with Redis we will need strongly typed access for persisting .NET objects and server side lists as .NET type IList<T>. Because of these requirements I chose ServiceStack.Redis [39] library. It support low level byte-access to Redis and also strongly typed high level access.

[7] LINQ stands for „language integrated query", it is a set of operators that add native quering capabilites into .NET languages [38]

4.4. User documentation

This subchapter shortly describes running the benchmark application from attachment CD.

4.4.1. Installing and running servers

- **.NET framework 4** – is required for benchmark application, just install redistributable package "Framework\dotNetFx40_Full_x86_x64.exe"
- **SQL Server 2008R2 Express** – Run "Servers\SQLEXPR32_x86_ENU.exe" from Attachments CD, follow instructions.
- **MongoDB** – Copy all files from "Servers\mongodb-win32-x86_64-1.8.2\bin" to "C:\data\db". Run file "mongod.exe"
- **Redis** – Run file "Servers\redis - 32bit\redis-server.exe"

4.4.2. Running benchmark

Run "BenchmarkApp\LauncherApp\bin\Release\LauncherApp.exe". Select number of clients, inserts and queries and hit the link of selected benchmark. Wait until every client is ready and then click on the "Release mutex" button to start the tests. Time count will be displayed in every client.

4.5. Platform specification

The platform used for these tests was a 2.80GHz Intel Core i7 with 8GB or memory and an SATA disk drive. The operating system was Windows 7 Professional.

4.6. Benchmark results

Here I will present results of all tests performed. Tables always show time needed for every query run and average time computed from 5 runs. Last column is computed amount of queries per second.

4.6.1. Basic Inserts

Table 2 – SQL Basic Inserts

Rows	time 1	time 2	time 3	time 4	time 5	**time avg.**	Q/sec
50	0,110	0,076	0,058	0,080	0,076	**0,080**	625
100	0,136	0,090	0,078	0,081	0,104	**0,098**	1 022
500	0,331	0,197	0,185	0,201	0,216	**0,226**	2 212
1 000	0,448	0,335	0,362	0,381	0,412	**0,388**	2 580
5 000	2,821	2,526	2,681	2,580	2,621	**2,646**	1 890
10 000	8,082	7,878	8,606	8,412	8,640	**8,324**	1 201
50 000	161,602	151,105	153,487	159,858	159,171	**157,045**	318
100 000	630,91	650,092	641,449	649,669	651,456	**644,7152**	155

Table 3 - MongoDB Basic Inserts

Rows	time 1	time 2	time 3	time 4	time 5	**time avg.**	Q/sec
50	0,007	0,007	0,007	0,007	0,007	**0,007**	7 143
100	0,010	0,009	0,011	0,009	0,010	**0,010**	10 204
500	0,014	0,012	0,017	0,019	0,012	**0,015**	33 784
1 000	0,064	0,017	0,019	0,018	0,017	**0,027**	37 037
5 000	0,160	0,084	0,201	0,139	0,177	**0,152**	32 843
10 000	0,269	0,274	0,213	0,286	0,235	**0,255**	39 154
50 000	1,497	1,294	1,265	1,265	1,126	**1,289**	38 778
100 000	3,182	3,020	2,936	2,945	2,856	**2,988**	33 469

Table 4 - Redis Basic Inserts

Rows	time 1	time 2	time 3	time 4	time 5	**time avg.**	Q/sec
50	0,002	0,002	0,002	0,001	0,001	**0,002**	31250
100	0,004	0,004	0,004	0,003	0,003	**0,004**	27778
500	0,015	0,017	0,017	0,018	0,018	**0,017**	29412
1 000	0,032	0,032	0,028	0,033	0,033	**0,032**	31646
5 000	0,176	0,177	0,181	0,165	0,172	**0,174**	28703
10 000	0,316	0,329	0,354	0,349	0,342	**0,338**	29586
50 000	1,784	1,804	1,676	1,737	1,623	**1,725**	28989
100 000	3,409	3,817	3,558	3,767	3,684	**3,647**	27420

As we can see I added on Table 2, Table 3 and Table 4 one largest dataset. It is because performance drop SQL Server showed on dataset larger than 50000. Also during the test with larger datasets SQL Server used nearly 100% of processor load. In my opinion this shows that Entity Framework has problems with large atomic inserts from multiple sources. So different methods should be used, e.g. buffering inserts and executing them in bulk by application (SQL Server has T-SQL statement for that called BULK INSERT, see [40]).

MongoDB performed more than 10 times faster and Redis was about 3 times faster than MongoDB. This in my opinion shows that MongoDB is very good for heavy write systems, and same with Redis – if we have enough main memory for it.

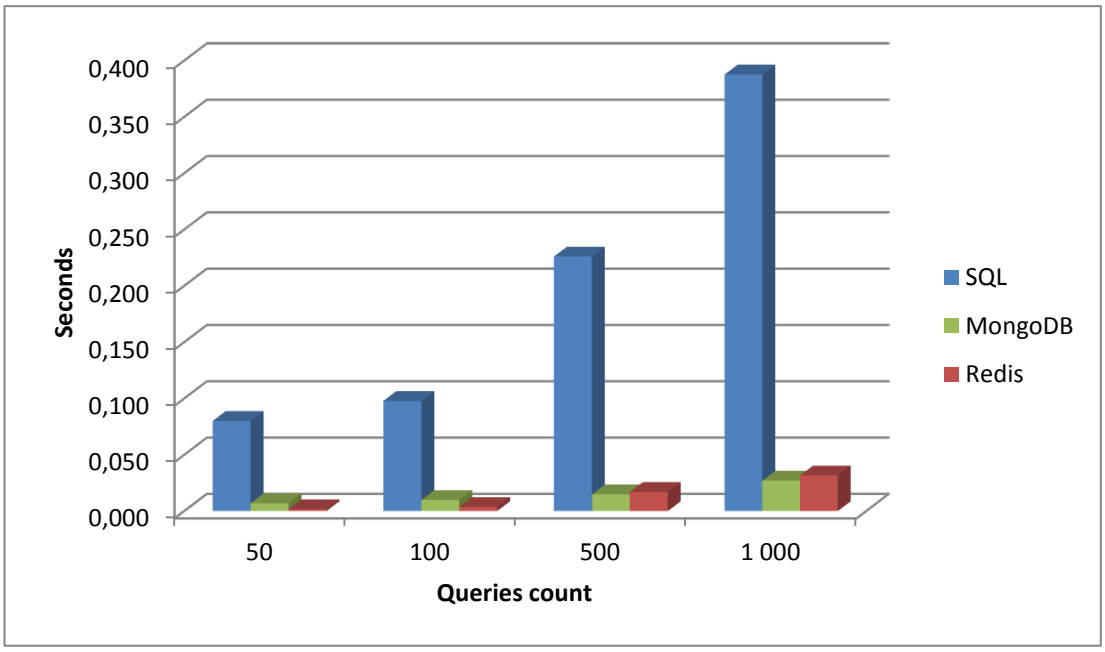


Figure 27 - Basic Inserts comparison graph (first part)

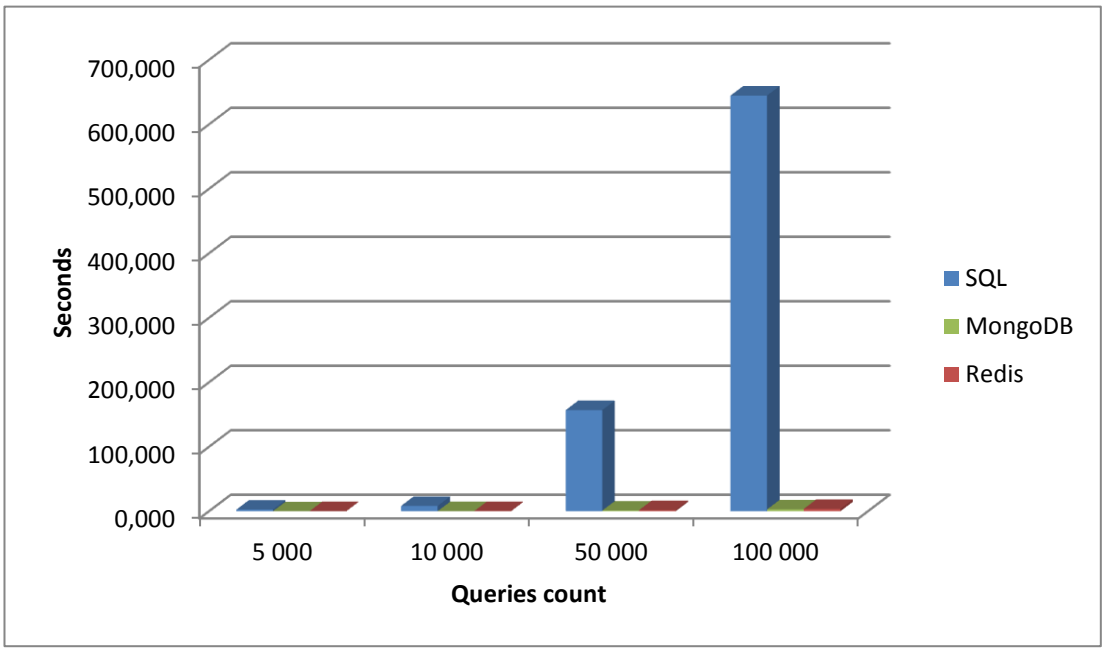


Figure 28 - Basic Inserts comparison graph (second part)

Graph for basic inserts speed comparison was split into two (Figure 27 and Figure 28) to demonstrate performance drop on SQL Server on large datasets.

4.6.2. Basic Search Queries on Primary Key

Table 5 - SQL Basic searches (Primary key)

Rows	time 1	time 2	time 3	time 4	time 5	time avg.	Q/sec
50	0,030	0,027	0,027	0,032	0,031	**0,029**	1701
100	0,109	0,120	0,125	0,109	0,113	**0,115**	868
500	0,170	0,194	0,182	0,154	0,170	**0,174**	2874
1 000	0,522	0,525	0,532	0,540	0,532	**0,530**	1886
5 000	1,889	1,858	1,749	1,852	1,869	**1,843**	2712
10 000	3,461	3,427	3,384	3,546	3,593	**3,482**	2872
50 000	15,022	16,630	15,969	16,232	16,792	**16,129**	3100

Table 6 - MongoDB Basic searches (Primary key)

Rows	time 1	time 2	time 3	time 4	time 5	time avg.	Q/sec
50	0,033	0,149	0,149	0,149	0,146	**0,125**	399
100	0,117	0,199	0,238	0,216	0,216	**0,197**	507
500	0,318	0,290	0,338	0,260	0,331	**0,307**	1627
1 000	0,398	0,334	0,379	0,415	0,445	**0,394**	2537
5 000	1,295	1,197	1,250	1,130	1,088	**1,192**	4195
10 000	2,384	2,115	2,160	2,160	2,059	**2,176**	4596
50 000	9,907	9,163	9,984	9,938	9,772	**9,753**	5127

Table 7 - Redis Basic searches (Primary key)

Rows	time 1	time 2	time 3	time 4	time 5	time avg.	Q/sec
50	0,095	0,107	0,101	0,047	0,095	**0,089**	562
100	0,077	0,155	0,123	0,116	0,13	**0,120**	832
500	0,243	0,189	0,197	0,115	0,217	**0,192**	2 601
1 000	0,117	0,281	0,258	0,272	0,293	**0,244**	4 095
5 000	0,505	0,446	0,557	0,458	0,588	**0,511**	9 789
10 000	0,947	0,604	0,729	0,649	0,616	**0,709**	14 104
50 000	3,072	3,289	3,146	3,057	2,702	**3,053**	16 376

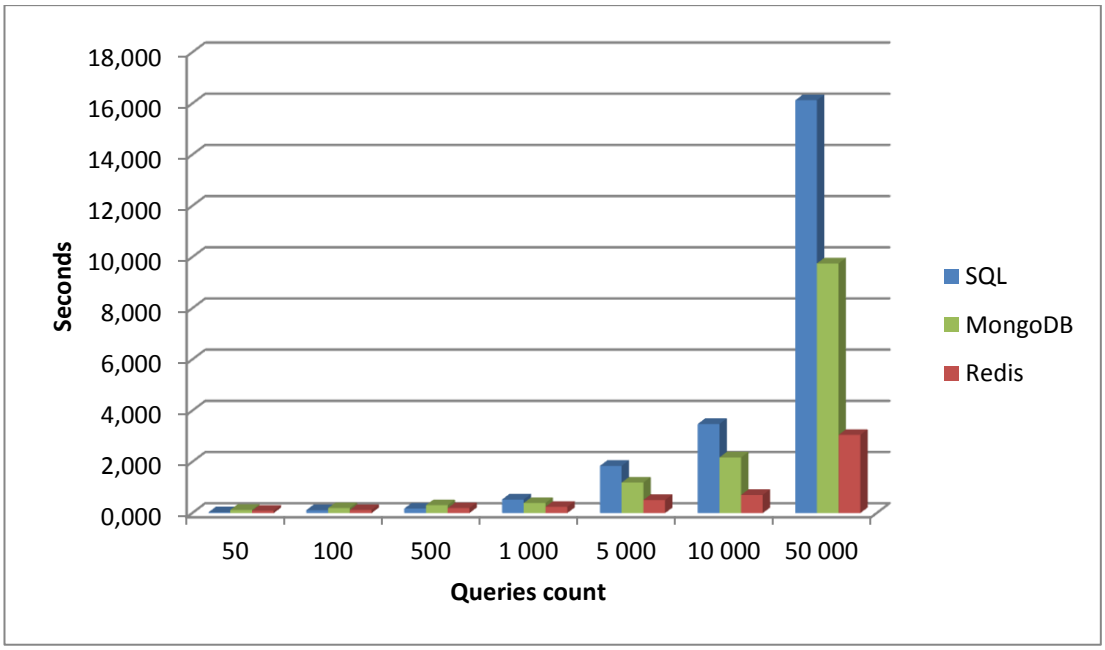


Figure 29 - Basic searches (Primary key) graph comparison

Table 5, Table 6, Table 7 and Figure 29 shows performance on basic searches with querying primary keys. SQL Server is little faster on small queries but on larger amount of queries is MongoDB nearly 2 times faster and Redis 5 times. No significant performance drops was detected, in my opinion all tested data stores are well read optimised.

4.6.3. Basic Search Queries with Update

Table 8 - SQL Basic Updates

Rows	time 1	time 2	time 3	time 4	time 5	**time avg.**	Q/sec
50	0,238	0,245	0,286	0,240	0,235	**0,249**	201
100	0,259	0,284	0,271	0,267	0,277	**0,272**	368
500	0,453	0,477	0,471	0,487	0,490	**0,476**	1051
1 000	0,712	0,686	0,670	0,669	0,624	**0,672**	1488
5 000	3,054	2,357	2,426	2,975	2,334	**2,629**	1902
10 000	7,184	4,404	4,549	4,540	4,449	**5,025**	1990
50 000	83,248	22,433	21,698	21,913	22,484	**34,355**	1455

Table 9 - MongoDB Basic Updates

Rows	time 1	time 2	time 3	time 4	time 5	**time avg.**	Q/sec
50	0,032	0,226	0,221	0,540	0,219	**0,248**	202
100	0,137	0,285	0,277	0,289	0,304	**0,258**	387
500	0,338	0,255	0,345	0,412	0,418	**0,354**	1414
1 000	0,433	0,447	0,506	0,567	0,494	**0,489**	2043
5 000	1,218	1,226	1,238	1,346	1,226	**1,251**	3997
10 000	2,108	2,218	2,224	2,283	2,200	**2,207**	4532
50 000	10,034	11,033	9,277	8,846	10,203	**9,879**	5061

Table 10 - Redis Basic Updates

Rows	time 1	time 2	time 3	time 4	time 5	time avg.	Q/sec
50	0,061	0,233	0,191	0,235	0,213	**0,187**	268
100	0,070	0,279	0,303	0,287	0,286	**0,245**	408
500	0,088	0,212	0,338	0,372	0,349	**0,272**	1 840
1 000	0,241	0,439	0,403	0,420	0,426	**0,386**	2 592
5 000	0,837	0,589	0,891	0,833	0,855	**0,801**	6 242
10 000	1,121	1,344	1,294	1,188	0,998	**1,189**	8 410
50 000	6,195	6,781	6,125	6,886	6,780	**6,553**	7 630

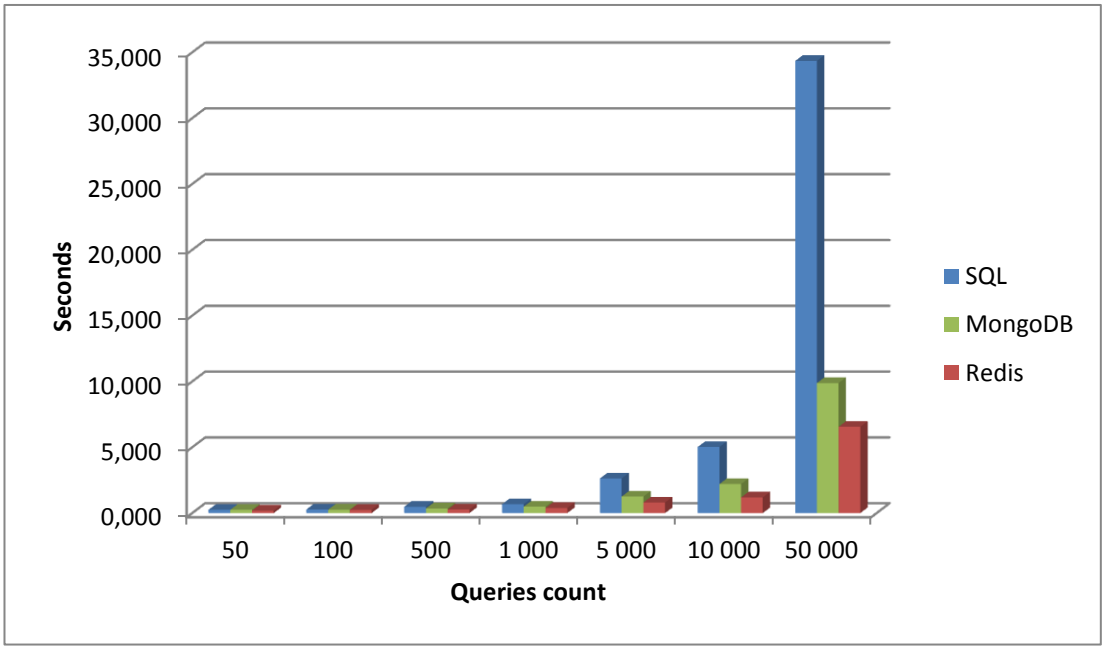


Figure 30 – Basic Updates comparison graph

Table 8, Table 9, Table 10 and Figure 30 shows performance on basic searches with querying primary keys and updating related value. Again there can be seen performance drop (compared to NoSQL solutions) on SQL Server on large datasets. Only way in my opinion how to overcome that is by bypassing ORM by direct T-SQL queries.

4.6.4. Basic Search Queries on Non-Indexed field

Table 11 - SQL Basic searches (non-index field)

Rows	time 1	time 2	time 3	time 4	time 5	**time avg.**	Q/sec
50	0,064	0,214	0,157	0,102	0,136	**0,135**	371
100	0,156	0,157	0,172	0,137	0,195	**0,163**	612
500	1,279	1,300	1,274	1,264	1,246	**1,273**	393
1 000	4,297	4,211	4,229	4,317	4,266	**4,264**	235
5 000	97,845	95,907	95,226	96,551	94,550	**96,016**	52
10 000	381,410	385,116	388,914	379,037	388,661	**384,628**	26

Table 12 - MongoDB Basic searches (non-index field)

Rows	time 1	time 2	time 3	time 4	time 5	**time avg.**	Q/sec
50	0,063	0,168	0,240	0,229	0,201	**0,180**	277
100	0,113	0,375	0,358	0,392	0,354	**0,318**	314
500	2,066	2,167	2,268	2,254	2,291	**2,209**	226
1 000	8,100	8,094	8,284	8,402	8,022	**8,180**	122
5 000	195,679	189,681	189,987	196,804	198,028	**194,036**	26
10 000	769,977	794,442	729,941	794,559	778,863	**773,556**	13

Table 13 - Redis Basic searches (non-index field)

Rows	time 1	time 2	time 3	time 4	time 5	**time avg.**	Q/sec
50	0,091	0,185	0,228	0,220	0,222	**0,189**	264
100	0,110	0,109	0,107	0,065	0,122	**0,103**	975
500	0,539	0,642	0,725	0,702	0,758	**0,673**	743
1 000	2,083	2,158	1,935	2,169	2,140	**2,097**	477
5 000	48,199	48,701	45,865	51,300	44,110	**47,635**	105
10 000	201,366	196,849	174,262	183,362	171,940	**185,556**	54

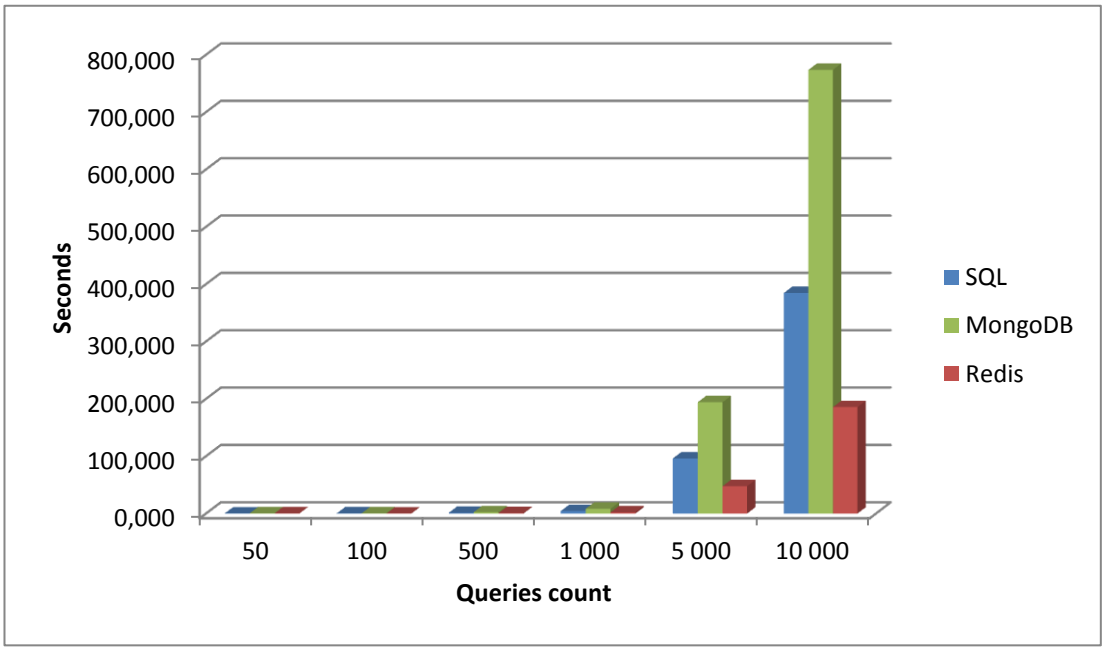


Figure 31 - Basic search queries on non-indexed field comparison graph

Table 11, Table 12, Table 13 and Figure 31 shows search queries comparison by string non-indexed field. As we can see I removed last largest 50000 rows big dataset. It is because very poor performance of every solution tested. Even fastest Redis needed about 50minutes at 100% processor use. Every solution experienced big performance drop at 5000 rows. SQL Server performed for the first time better than MongoDB. In my opinion this test showed that querying non-indexed string field is really worst case scenario for all solutions tested.

4.6.5. Complex Queries

Table 14 - SQL Complex queries

Rows	time 1	time 2	time 3	time 4	time 5	time avg.	Q/sec
50	0,306	0,288	0,309	0,300	0,318	**0,304**	164
100	0,381	0,359	0,346	0,388	0,305	**0,356**	281
500	0,911	1,001	1,004	0,941	0,956	**0,963**	519
1 000	1,890	1,804	1,742	1,857	1,812	**1,821**	549
5 000	11,762	12,001	11,859	11,920	11,522	**11,813**	423
10 000	33,273	33,100	32,436	33,043	33,596	**33,090**	302
50 000	288,718	286,617	286,111	289,261	292,844	**288,710**	173

Table 15 - MongoDB Complex queries

Rows	time 1	time 2	time 3	time 4	time 5	time avg.	Q/sec
50	0,067	0,061	0,286	0,244	0,295	**0,191**	262
100	0,092	0,364	0,316	0,307	0,382	**0,292**	342
500	0,347	0,419	0,341	0,395	0,346	**0,370**	1353
1 000	0,493	0,495	0,610	0,575	0,572	**0,549**	1821
5 000	1,652	1,657	1,851	1,853	1,534	**1,709**	2925
10 000	3,368	3,900	3,090	3,613	3,434	**3,481**	2873
50 000	31,691	30,916	32,680	32,891	33,286	**32,293**	1548

Table 16 - Redis Complex queries

Rows	time 1	time 2	time 3	time 4	time 5	time avg.	Q/sec
50	0,182	0,080	0,255	0,301	0,304	**0,224**	223
100	0,095	0,344	0,336	0,338	0,351	**0,293**	342
500	0,148	0,103	0,205	0,213	0,345	**0,203**	2 465
1 000	0,106	0,425	0,389	0,286	0,405	**0,322**	3 104
5 000	0,150	0,087	0,311	0,402	0,423	**0,275**	18 208
10 000	0,312	0,211	0,453	0,460	0,507	**0,389**	25 733
50 000	4,630	4,863	4,792	4,800	4,621	**4,741**	10 546

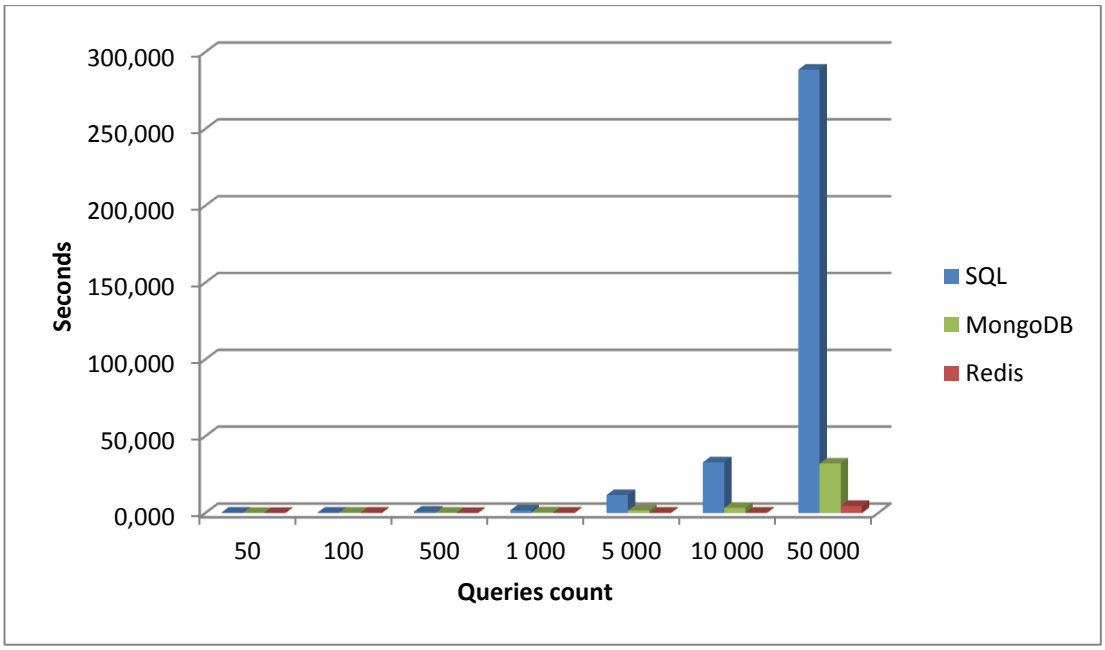


Figure 32 - Complex queries comparison graph

Table 14, Table 15, Table 16 and Figure 32 shows results of complex query test on object Customer and his two subsets. Entity Framework by my experience keeps sub-objects in different tables and connects them to main object by foreign keys – that means a lot of queries when user wants them all. Because of that I was expecting that NoSQL solutions with theirs included sub-object will perform way better. On the largest dataset was MongoDB nearly 10 times faster than SQL Server and Redis nearly 10 times faster than MongoDB. In my opinion it shows clearly advantage of embedded sub-objects.

4.7. Benchmark Conclusion

It is not a big surprise that Redis performed best in every test. Mongo always performed more than 2-times faster than SQL Server with one exception. MongoDB is 2-times slower on queries on non-indexed fields, I wasn't able to find exact reason but probably it's just temporary problem – MongoDB development is very agile. When we will use only SQL difference will be even larger. In next test should be examined Mongo MapReduce querying capabilities. Also based on experiences I get from this benchmark I would create more robust and universal object model for next application.

I hope that this simple benchmark can help developers with finding the right tools for their applications and it demonstrated basic differences between designing persistent objects in NoSQL and SQL.

Conclusion

This thesis certainly does not cover every use case, benefit or drawback of SQL and NoSQL, but I think it gives a pretty decent start. Purpose of this thesis was not to convince all readers that we should throw away all our relational databases and replace them with NoSQL. It was to help readers to understand alternatives to relational databases, their advantages and disadvantages and that they exist for the reason; we can picture our data problems and their solutions as Figure 33.

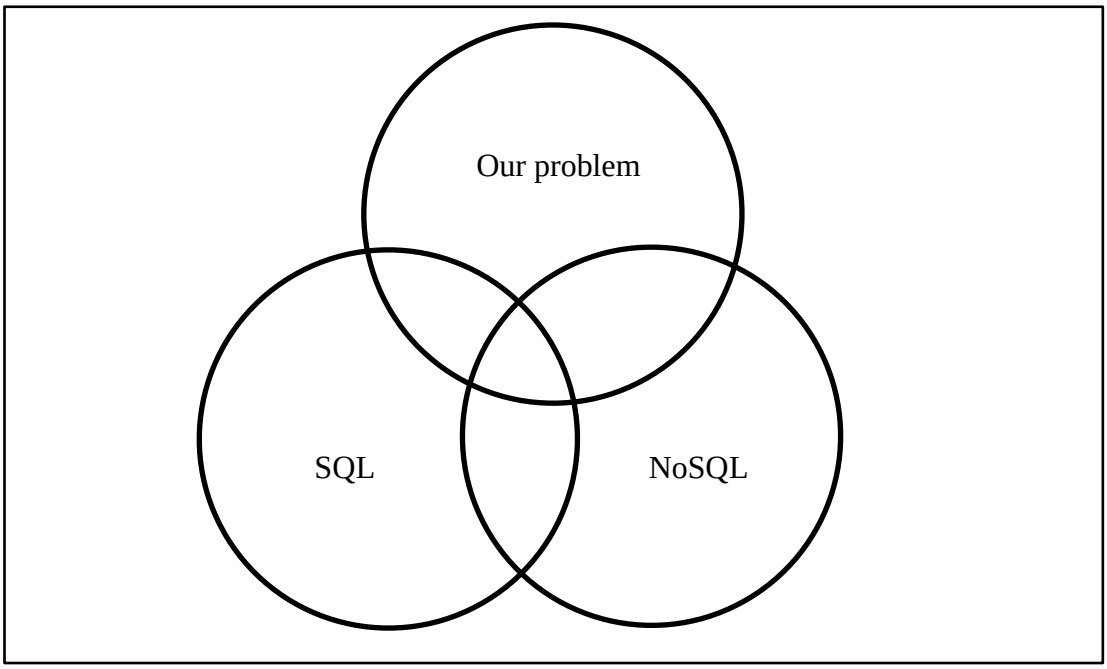


Figure 33 – Venn diagram of SQL and NoSQL solutions

In my opinion when our data can fit into standard RDBMS without too many compromises we do not need NoSQL. E.g. if classic size MySQL server fits our needs it is probably what we needs. If our entities are homogenous and simple we will have no problem mapping them on tables. This is what RDBMS was doing for decades and where really shines. But when it comes to distribution and scaling we should really take a step back and select not just default tool, but best tool for our use case.

Bibliography

1. **IDC.** The Expanding Digital Universe. [Online] March 2007. [Cited: 12 12 2010.] http://www.emc.com/collateral/analyst-reports/expanding-digital-idc-white-paper.pdf.
2. **WEBAXES.** How to Improving Database Performance with Partitioning. [Online] http://www.webaxes.com/2010/07/how-to-improving-database-performance-with-partitioning/.
3. **Gray, Jim, et al.** The Dangers of Replication and a Solution. [Online] 26 5 1996. [Cited: 5 5 2011.] http://research.microsoft.com/~gray/replicas.ps.
4. **Eifrem, Emil.** NoSQL and Graph Databases. [Online] 2010. [Cited: 1 2 2011.] http://www.slideshare.net/emileifrem/nosql-overview-neo4j-intro-and-production-example-qcon-london-2010.
5. **Graham, Paul.** Web 2.0. [Online] 11 2005. [Cited: 12 5 2011.] Paul Graham.
6. **Gray, Jim.** The Transaction Concept: Virtues and Limitations. [Online] June 1981. [Cited: 1 5 2011.] http://research.microsoft.com/en-us/um/people/gray/papers/theTransactionConcept.pdf.
7. **Hohpe, Gregor.** Starbucks Does Not Use Two-Phase Commit. [Online] 2004. [Cited: 21 12 2010.] http://www.eaipatterns.com/ramblings/18_starbucks.html.
8. **Duxbury, Bryan.** Matching Impedance: When to use HBase. [Online] 11 3 2008. [Cited: 3 2 2011.] http://blog.rapleaf.com/dev/2008/03/11/matching-impedance-when-to-use-hbase/.
9. **Hewitt, Eben.** *Cassandra: The Definitive Guide.* s.l. : O'Reilly, 2010.
10. **Callaghan, Mark.** The futures of replication in MySQL. [Online] 21 8 2009. [Cited: 11 5 2011.] https://www.facebook.com/note.php?note_id=126049465932.
11. **Persyn, Jurriaan.** Database Sharding at Netlog, with MySQL and PHP. [Online] 12 2 2009. [Cited: 3 1 2011.] http://www.jurriaanpersyn.com/archives/2009/02/12/database-sharding-at-netlog-with-mysql-and-php/#implications.
12. **Oracle.** Using Replication for Scale-Out. [Online] 2011. http://dev.mysql.com/doc/refman/5.1/en/replication-solutions-scaleout.html.
13. **Owens, Michael.** *The Definitive Guide to SQLite.* s.l. : Apress, 2006. ISBN 978-1-59059-673-9.
14. sqlite - Speed Comparison. [Online] 15 12 2007. [Cited: 2 5 2011.] http://www.sqlite.org/cvstrac/wiki?p=SpeedComparison.
15. NHibernate Reference Documentation. [Online] http://www.nhforge.org/doc/nh/en/index.html#architecture-overview.
16. **ORMBattle.NET Team.** ORM Comparison and Benchmarks on ORMBattle.NET. [Online] 30 7 2010. [Cited: 5 5 2011.] http://ormbattle.net/.
17. NOSQL meetup. [Online] http://nosql.eventbrite.com/.
18. **Edlich, Stefan.** NOSQL Databases. [Online] 2011. [Cited: 4 1 2011.] http://nosql-database.org/.
19. **Brewer, Eric.** Towards Robust Distributed Systems. [Online] 19 7 2000. [Cited: 9 1 2011.] http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf.
20. **Nancy Lynch, Seth Gilbert.** Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. [Online] 2002. [Cited: 5 5 2011.] http://lpd.epfl.ch/sgilbert/pubs/BrewersConjecture-SigAct.pdf.
21. **Vogels, Werner, et al.** Dynamo: Amazon’s Highly Available Key-value Store. [Online] 2 10 2007. [Cited: 5 1 2011.] http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html.

22. **Gupta, Vineet.** NoSql Databases – Part 1 - Landscape . [Online] 5 1, 2010. [Cited: 2 5, 2011.] http://www.vineetgupta.com/2010/01/nosql-databases-part-1-landscape/.
23. **Czajkowski, Grzegorz.** Sorting 1PB with MapReduce. [Online] 21 11 2008. [Cited: 1 3 2011.] http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html.
24. **Caraciolo, Marcel.** MapReduce with MongoDB and Python. [Online] 21 8 2010. [Cited: 5 5 2011.] http://aimotion.blogspot.com/2010/08/mapreduce-with-mongodb-and-python.html.
25. **Bain, Tony.** Is the Relational Database Doomed? [Online] 9 2 2009. [Cited: 16 12 2010.] http://www.readwriteweb.com/enterprise/2009/02/is-the-relational-database-doomedp2.php.
26. **Nakashima, Jim.** Walkthrough: Windows Azure Table Storage (Nov 2009 and later). [Online] 2010. [Cited: 12 12 2010.] http://blogs.msdn.com/b/jnak/archive/2008/10/28/walkthrough-simple-table-storage.aspx.
27. **Sanfilippo, Salvatore.** Redis Clients. [Online] 2 4 2011. [Cited: 14 5 2011.] http://redis.io/clients.
28. —. Redis Features. [Online] 23 12 2009. [Cited: 12 12 2010.] http://code.google.com/p/redis/wiki/Features.
29. **Springer, Wilfred.** nosql rollercoaster. [Online] 6 11 2010. [Cited: 12 12 2010.] http://www.slideshare.net/springerw/nosql-rollercoaster.
30. **Merriman, Dwight.** mongoDB. [Online] 20 11 2009. [Cited: 12 12 2010.] http://assets.leadit.us/mysql/MongoDB-10gen-CEO-Dwight-Merriman-presenting-at-NYC-MySQL-Group-at-Sun-Microsystems.pdf .
31. **Chang, Fay, et al.** Bigtable: A Distributed Storage System for Structured Data. [Online] 11 2006. [Cited: 3 1 2011.] http://labs.google.com/papers/bigtable.html.
32. [Online] http://nosql.mypopescu.com/post/1611716462/hbase-at-facebook-and-why-not-mysql-or-cassandra.
33. **Borthakur, Dhruba.** HDFS Architecture. [Online] 19 2 2010. [Cited: 12 4 2011.] http://hadoop.apache.org/common/docs/r0.20.2/hdfs_design.html#Replication+Pipelining.
34. **Peschka, Jeremiah.** Facebook Messaging – HBase Comes of Age. [Online] 10 11 2010. [Cited: 15 1 2011.] http://facility9.com/2010/11/18/facebook-messaging-hbase-comes-of-age.
35. **Kennedy, Michael.** MongoDB vs. SQL Server 2008 Performance Showdown. [Online] [Cited: 11 07 2011.] http://www.michaelckennedy.net/blog/2010/04/29/MongoDBVsSQLServer2008PerformanceShowdown.aspx.
36. **Lerman, Julia.** *Programming Entity Framework.* Sebastopol : O'Reilly, 2010. ISBN 978-0-596-80726-9.
37. NoRM is a .Net library for connecting to the document-oriented database, MongoDB. [Online] [Cited: 12 7 2011.] http://normproject.org/.
38. Object IDs. *MongoDB manual.* [Online] [Cited: 18 7 2011.] http://www.mongodb.org/display/DOCS/Object+IDs.
39. **Bellot, Demis.** The ServiceStack.Redis C# Client Library. *GitHub.* [Online] [Cited: 7 7 2011.] https://github.com/ServiceStack/ServiceStack.Redis.
40. **Microsoft.** BULK INSERT (Transact-SQL). *MSDN Library.* [Online] [Cited: 11 7 2011.] http://msdn.microsoft.com/en-us/library/ms188365.aspx.

41. **Wood, John.** CouchDB: Databases and Documents. [Online] 30 6 2009. [Cited: 1 12 2010.] http://johnpwood.net/2009/06/30/couchdb-databases-and-documents/.
42. **Microsoft.** Mutex Class. *MSDN Library.* [Online] [Cited: 10 7 2011.] http://msdn.microsoft.com/en-us/library/system.threading.mutex.aspx.

List of Tables

Attachments

Attachments - CD

Benchmark (folder) – source codes of BechmarkApp + database servers' installation files

Thesis.pdf – electronic version of thesis